Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

A language to query execution traces of Java programs

Verhoustraeten, Ludovic

Award date:
2016

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. May. 2023

A language to query execution traces of Java programs

Ludovic Verhoustraeten
Université de Namur,
Namur,
Belgique,
lverhoustraeten@gmail.com

25 août 2016

REMERCIEMENTS

En préambule à ce mémoire, je souhaite adresser mes remerciements les plus sincères aux personnes qui m'ont apporté leur aide et qui ont contribué à l'élaboration de mon travail. Je tiens à remercier tout particulièrement Dr. V.Englebert, qui en tant que promoteur, s'est toujours montré à l'écoute et disponible tout au long de la réalisation de ce mémoire. Lorsque j'ai commencé, j'avais très peu de connaissances sur le 'Code Querying', mais M. Englebert a toujours pris le temps de partager avec moi son expertise et sa vision à propos de mon sujet de mémoire. Finalement, je voudrais remercier mes amis et ma famille, en particulier mon père qui m'a toujours soutenu et encouragé durant cette longue route.

ABSTRACT

The job of a developer often boils down to write and give structure to the code, but few of them leave a source of documentation to understand this structure in its maintenance phase. The person in charge of this maintenance then fits into the role of an explorer in search of such information. During this research, he will answer his questions using tools that allow him to better respond to his questions. There are a multitude of tools that uses query languages to query the source code. In addition to these tools, which are mainly made for static queries, there are many techniques to collect execution traces. This thesis proposes to implement a query language for querying the execution traces (from Java programs), as query tools for source code do. In this thesis, we will begin by a review of the literature ("Querying Code") in order to have an overview of existing languages and their weaknesses. Then, based on this research and through an incremental approach, we imagined, created and tested our own language. During the test phase, we developed a script of questions to test our theoretical choice of composition and for testing the scalability of this language. Our approach shows that our language is able to incorporate new features without having to change our founding bases. The results also highlight the importance of the relationship between then static and dynamic universes of a program during the questioning phase of a program. We showed that it's possible, using the right approach, to create a query language binding the dynamic universe from execution traces to a static universe and therefore obtain a good level of expressiveness.

Keywords : Code Querying, execution trace, query language.

Le travail d'un développeur se résume souvent à écrire et structurer du code, mais peu d'entre eux laissent une source de documentation afin de comprendre cette structure lors de sa phase de maintenance. La personne en charge de cette maintenance rentre alors dans un rôle d'explorateur partant à la recherche de ces informations. Durant cette recherche, il voudra répondre aux questions qu'il se pose en utilisant des outils qui lui permettent de répondre au mieux à ses questions. Il existe une multitude d'outils qui utilise des langages de requête pour questionner du code source. En plus de ces outils, qui répondent principalement à des interrogations d'ordre statique, il existe de nombreuses techniques pour récolter des traces d'exécution. Ce mémoire propose de mettre en place un langage de requête permettant d'interroger ces traces d'exécution (issues de programme Java), comme les font les outils d'interrogation de code source.

Dans ce mémoire, nous procéderons en 1er lieu par une analyse de la littérature (« Code Querying ») afin d'avoir une vue d'ensemble des langages existants et leurs faiblesses. Ensuite, sur base de cette recherche et grâce à une approche incrémentale, nous avons imaginé, créé et testé un langage. Durant la phase de test, nous avons dressé un scénario de questions pour mettre à l'épreuve nos choix théoriques de composition et pour tester la capacité d'évolution de ce langage. Notre démarche a montré que notre langage est capable d'intégrer de nouvelles fonctionnalités sans devoir changer nos bases fondatrices.

Les résultats soulignent également l'importance du lien qui existe entre l'univers statique et dynamique d'un programme lors de la phase de questionnement d'un programme. Nous avons montré qu'il est possible, en utilisant une bonne approche, de créer un langage de requête liant l'univers dynamique issu de traces d'exécution à un univers statique et dès lors, obtenir un bon niveau d'expressivité.

Mots-clefs : Code Querying, trace d'exécution, langage de requête.

Table des matières

Table des figures

1. INTRODUCTION GÉNÉRALE

1.1 Contexte

La compréhension d'applications dans le contexte de la maintenance informatique est un sujet largement discuté et ayant suscité la réalisation de nombreuses recherches. En effet, maintenir une application représente une grosse partie dans le cycle de vie d'un logiciel, pouvant engendrer des coûts importants pour le projet. Et ce n'est pas sans raison : cette tâche se révèle généralement comme étant un travail bien fastidieux à accomplir.

Les Systèmes se créent autour d'idées parfois confuses, évoluant avec l'arrivée et le départ des différents développeurs qui laissent souvent leur code comme seule source de documentation fiable. Avec les années, les applications deviennent de plus en plus complexes, rendant la tâche des nouveaux spécialistes de plus en plus difficile. Dans certains cas extrêmes, il n'existe parfois plus aucune personne de contact pouvant renseigner sur les secrets architecturaux et sur les comportements du programme.

Dans le monde de la programmation, on nous répète souvent :"si après avoir lu uniquement les commentaires d'un programme vous n'en comprenez pas le fonctionnement, jetez le tout !" La réalité n'est malheureusement pas aussi simple : nous ne pouvons pas toujours mettre à la poubelle une application en production représentant des années de développement.

Le travail du développeur devient alors comme un jeu de piste en pleine jungle où il prend le rôle d'un explorateur se trouvant seul à la croisée des chemins sans carte ni boussole, où seuls l'instinct et l'expérience permettent de prendre un chemin plutôt qu'un autre. Il doit se lancer à l'aventure en ouvrant des fichiers et en parcourant le code pour tenter petit à petit de dresser une carte cohérente du programme.

Cette aventure n'est généralement pas sans embûches : la découverte d'un nouveau module ne révèle pas toujours ses secrets au premier coup d'œil, certaines parties ne sont parfois présentes que pour un cas bien particulier, ou tout simplement inutilisées, n'étant plus que des vestiges du passé. Nous devons encore ajouter une dimension dynamique à l'exploration durant laquelle le développeur doit faire face à des éléments inconnus interagissant entre eux par des techniques de construction bien particulières qu'il n'arrive pas toujours à identifier clairement.

Ainsi, le travail de questionnement est particulièrement important dans le processus de recherche car il permet de confirmer des hypothèses et de se créer petit à petit une représentation mentale structurée du code. Mais c'est avant tout ce travail qui demande un effort cognitif plus ou moins conséquent afin d'associer à une question mentale un processus de recherche réalisable. C'est l'effort intellectuel qui permet au développeur de trouver la suite des portes à franchir pour trouver la réponse.

Ces portes d'accès sont généralement définies par l'environnement utilisé par le développeur et elles dépendent des fonctionnalités offertes par celui-ci. S'agit-il d'un simple traitement de texte qui n'offre pas un grand nombre de processus de recherche ou d'un IDE plus complexe offrant un

module de recherche plus puissant ?

C'est dans ce cadre que viennent s'inscrire les outils d'aide à la compréhension de programmes, tels que les outils de "code querying", ou des outils plus spécifiques, comme ceux qui récoltent des traces lors de l'exécution d'un programme. Ces outils visent à faciliter cette tâche de recherche en proposant des fonctionnalités répondant au mieux aux questions que les développeurs se posent. En leur diminuant ainsi l'effort cognitif à fournir, ils leur permettent de gagner un temps considérable.

1.2 Contributions

Ce mémoire se propose de trouver un langage de requête qui permet de faciliter la recherche d'informations parmi des traces d'exécution récoltées. Un peu comme le font les outils de "code querying" pour retrouver de l'information dans du code source. Pour y arriver, nous avons mené une analyse de la littérature pour cerner les différentes possibilités de langages existantes et nous en avons retiré les différentes remarques importantes. Nous avons par la suite dressé un scénario avec une liste de questions pertinentes pour valider et mettre à l'épreuve les choix théoriques issus de cette analyse initiale.

1.3 Structure du mémoire

Soucieux de bien éclairer le lecteur sur ce qui est traité dans notre mémoire et afin de lui en faciliter la lecture, nous commencerons par lui présenter ce qui se cache derrière la notion de recherche d'informations et nous définirons quelques termes en rapport avec ce domaine.

Nous sortirons ensuite de ce cadre purement théorique pour nous rapprocher le plus possible de notre domaine d'application en analysant des outils de "code querying". Bien que ceux-ci s'intéressent principalement (mais pas exclusivement) à l'analyse statique d'un programme, cela nous semble être une bonne approche. En effet, il est intéressant de profiter de l'expérience acquise dans la réalisation d'outils dont l'interface utilisateur fonctionne par un système de requêtes et dont le but est similaire au nôtre, à savoir, faire ressortir de l'information d'un "programme". Nous tenterons de présenter une vue générale des différents langages utilisés. Nous relèverons les différentes remarques émises par les concepteurs et nous listerons une série de cas d'utilisation.

Nous nous rapprocherons par la suite de l'aspect dynamique du problème en présentant les différences qui existent entre d'un coté, la compréhension d'un programme par l'analyse statique, et de l'autre, par l'analyse dynamique qui nous concerne plus particulièrement. Nous poursuivrons par une brève introduction aux techniques existantes de récolte de traces d'exécution et nous continuerons en tentant de donner une vision plus pragmatique en listant une série de besoins et de cas d'utilisation.

Ensuite, viendra la seconde partie de notre contribution où nous commencerons par choisir un langage de référence sur base de nos recherches effectuées, pour continuer par une mise à l'essai de nos choix avec un scénario d'utilisation.

Nous finirons cette partie par une conclusion présentant les limites de notre approche, les problèmes rencontrés et nous partagerons avec le lecteur les résultats de notre mise à l'essai.

2. INTRODUCTION À LA TERMINOLOGIE

2.1 Interrogation d'information

Avant de définir un "Query Langage" pour interroger nos traces d'exécution, nous devons comprendre les principes de base de la recherche d'informations. Il existe en effet 2 matières principales dans ce domaine : la représentation (ou modélisation) d'un corpus d'une part et l'interrogation du fonds documentaire ainsi constitué d'autre part. Nous y ferons référence au moment de la définition des concepts et nous les situerons dans le processus de recherche.

Définition

L'association des professionnels de l'information (ADBS) définit la recherche d'informations de la sorte : "Ensemble des méthodes, procédures et techniques ayant pour objet d'extraire d'un document ou d'un ensemble de documents les informations pertinentes".

Cette définition peut sembler très générale au premier abord, ne nous laissant pas le moindre indice pour une visualisation des techniques et processus utilisés. C'est tout à fait normal, car la recherche d'informations n'est pas seulement liée aux systèmes informatisés tels que nous pourrions être tentés de le penser : en fait, la recherche d'informations est une activité relativement ancienne déjà bien présente dans les bibliothèques. L'accélération de son étude coïncide avec l'apparition de l'informatique, il y a quelques décennies, notamment avec la création de SGBD et l'avènement du web. Il existe au moins autant de techniques, méthodes et procédures qu'il y a de types de "Systèmes de recherche".

Pour ce qui nous intéresse, nous pouvons modéliser globalement le processus de recherche de la sorte : Un usager ayant un besoin d'information adresse, généralement sous la forme d'une requête, une question à un système souvent à l'aide de mots-clés. Dans le système interrogé, les documents ont des représentations et sont indexés. Entre la requête et l'index du système s'opère alors un appariement qui fournira une réponse appelée "réponse pertinente". [23]

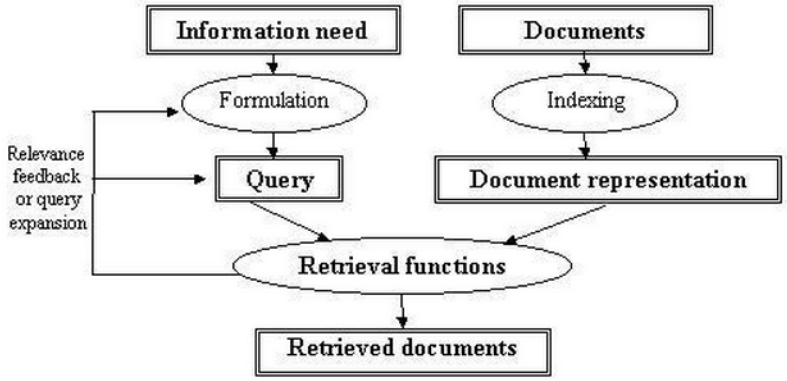
Les techniques de représentation et d'indexation sont largement utilisées dans les systèmes de recherche d'informations (SRI) modernes et font partie de ce que nous appelons la phase de prétraitement du corpus de documents [20].

La figure 2.1 représente le fonctionnement d'un système de recherche moderne [19]

2.2 Concept de requête

Maintenant que nous avons présenté le processus de recherche de manière générale, il est important de comprendre la terminologie qui tourne autour du concept de requête. Dans notre précédent exemple, nous avons fait référence à de simples mots-clés que l'utilisateur pourrait employer afin de retrouver de l'information notamment grâce à la mise en place d'un index. Dans la réalité,

Fig. 2.1: Processus de récupération d'informations.



et nous le verrons dans le chapitre consacré au "Code Querying", une requête peut-être exprimée de bien des manières, allant d'un langage naturel très permissif à un langage de requête plus structuré comme le SQL. Dans le cadre de son livre "DSL Engineering" [32], Markus Voelter va plus loin en parlant de langages textuels, langages tabulaires et de langages graphiques.

Dans la suite de ce travail, nous nous sommes intéressés à la composition et à la découverte de langage de requête de type textuel. D'un point de vue de leurs descriptions, il est généralement de coutume de formaliser leur syntaxe de manière minutieuse par une grammaire BNF (Backus-Naur Form) ou un dérivé de cette grammaire. Nous ne ferons pas de présentation détaillée des différentes grammaires dans le cadre de ce mémoire.

2.3 Domain specific langage

Il est difficile de trouver une description précise et claire de ce que peut être un DSL (Domaine Specific Langage). Les définitions sont souvent faites dans le cadre d'exemples concrets très spécifiques, comparant des DSL à des langages de programmation traditionnels tels que le java ou le C. Mais nous découvrons dans la littérature également la présence des termes 'DSL' et 'DSL-like' pour décrire des langages de requêtes spécifiques [30] [8]. Cela dit, nous pouvons dire intuitivement qu'un langage dédié est conçu pour répondre aux préoccupations et contraintes d'un domaine d'application précis et qu'ils diffèrent des autres langages plus généralistes qui cherchent à répondre aux besoins d'un ensemble de domaines. L'idée générale de définir un tel langage est de simplifier l'effort à fournir par les utilisateurs en mettant à leur disposition des mots, opérateurs et abstractions faisant directement référence à des concepts d'un domaine cible.

La mise en place d'un tel langage peut s'avérer plus ou moins complexe en fonction de la technique utilisée.

Nous distinguerons 2 types de DSL [32], les DSL internes et externes. Les DSL internes sont généralement considérés comme des extensions d'un langage de programmation existant dont ils sont fortement dépendants. Les DSL externes sont à l'inverse des langages indépendants de tout langage existant possédant leurs propres parseur et compilateur syntaxiques. Dans ce dernier cas, la bonne pratique tend à mettre en place une grammaire 'BNF' pour formaliser l'aspect syntaxique du langage. Il est important de prendre en compte cette remarque, car le choix technique peut in-

fluencer fortement l'aspect de la syntaxe du langage proposé d'une part et peut, d'autre part, avoir également un impact important sur les possibilités d'évolution et d'optimisation du langage.

Pour donner un premier exemple, voici un DSL interne dit "SQL-like" défini en java :

```
SELECT().from(t).where(row(t.a, t.b).eq(1, 2)) ;
```

Un lecteur avisé remarquera que nous retrouvons une syntaxe bien marquée par le langage java. Ce phénomène de dépendance peut bien entendu s'atténuer en fonction du langage de développement utilisé, comme par exemple avec un langage où le "sucre syntaxique" est moins présent tel que le Scala. L'étude des outils de "code querying" fournit une meilleure vue d'ensemble des différents langages utilisés dans le domaine de l'interrogation de programme.

3. MÉTHODOLOGIE

D'un point de vue méthodologique, selon Markus Voelter [32], la meilleure approche dans la construction d'un DSL serait une approche itérative par l'application de tests. L'idée générale de cette approche est de construire progressivement le langage par l'ajout de cas d'utilisation au sein d'un scénario. Selon lui, il est également important de se familiariser avec le domaine d'application avant de passer à la réalisation.

Dans le cadre de ce mémoire, nous allons nous familiariser avec le domaine d'application et les différents langages que nous pourrions utiliser au travers d'une étude des outils de "Code Querying". Nous nous focaliserons ensuite sur l'aspect dynamique en présentant des cas d'instrumentation relatifs à la compréhension de programmes afin de pouvoir cerner avec plus de précision le genre de concept et d'informations que nous devrons manipuler.

4. CODE QUERYING

Dans ce chapitre, nous ferons une présentation de différents types de systèmes de recherche existants dans le domaine de l'interrogation de programmes basés sur un langage de requête. Il en existe de différents types, mais nous présenterons les plus pertinents d'entre eux regroupés par langage. Cela nous permettra de prendre connaissance des avantages et limites de chacune des approches.

Les outils de "code querying" sont des systèmes de recherche d'informations appliqués au domaine de l'interrogation de code source.

Le principe de requête sur des traces d'exécution, quant à lui, diffère dans son concept du "code querying" car il se concentre sur l'enregistrement d'événements au cours d'un processus d'exécution du code interprété ou compilé bien que son objectif reste fondamentalement similaire : "Faire ressortir de l'information d'une application au travers d'un système de requêtes qui peut exprimer des concepts communs de programmation tel que des appels de méthodes, de constructeurs, lecture d'une variable...".

Beaucoup d'outils de "code querying" reposent sur des langages d'interrogation spécifiques, mais qui restent largement inspirés ou dépendants des langages de manipulation de SGBD ou de la programmation logique. Mais il existe également d'autres approches notamment basées sur de l'algèbre [27] ou sur du langage naturel [21].

Pour notre présentation, nous avons créé plusieurs grandes familles :Les awk-Like, les SQL-Like (OMEGA [22]),les Prolog-Like(SOUL [12] [4], Jquery [7] [13] [6]), les DataLog-Like (CodeQuest [16] [18] [17]) plus une série d'autres approches telles que les langages de type graphe, algébrique et naturel.

4.1 AWK-Like

Une des premières approches dans le domaine de l'analyse de programme a été l'utilisation de scripts "awk" [16] basés sur un ensemble d'expressions régulières liées à du code impératif. Dans ce mécanisme, à chaque fois qu'un mot du code source analysé est trouvé par une expression régulière, des instructions sont exécutées [16]. Il peut s'agir d'un simple affichage d'informations sur la sortie standard tout comme des transformations directement effectuées dans le code source du programme.

Le principal avantage de cette technique est qu'elle est simple à mettre en œuvre et qu'elle n'est pas dépendante d'un langage en particulier. Elle ne demande pas non plus une représentation du code source, tel qu'un arbre syntaxique abstrait (AST).

La limite de ce genre d'outils, selon Linton [22], est qu'ils ne travaillent qu'avec des notions textuelles telles que des mots, des caractères, des noms de fichiers et des lignes. Toute correspondance avec des entités de programmation permettant par exemple de reconnaître des modules et/ou des variables s'avèrerait difficile, voire impossible. En tout cas pas autrement qu'avec des conventions de nommage. Toute interprétation syntaxique du code source serait donc beaucoup trop coûteuse par ce procédé.

4.2 SQL-Like

OMEGA

Linton avance en 1983 l'environnement OMEGA [22], qui propose de représenter un programme comme une base de données relationnelle. Pour y arriver, il définit un ensemble de 58 relations essentielles qui permettent la modélisation. Son environnement est particulièrement intéressant, car il permet d'exploiter conjointement l'information statique issue du code source et l'information dynamique issue de l'exécution de celui-ci. L'environnement OMEGA est représenté dans la figure 4.1 en page 17 :

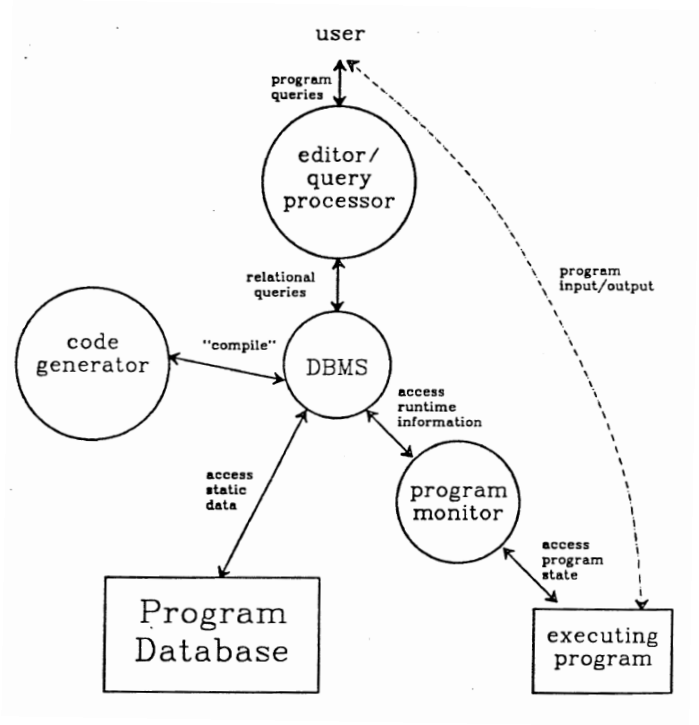


Fig. 4.1: Structure de l'environnement d'OMEGA

Le code source des programmes analysés est entreposé dans une base de données relationnelle et l'information dynamique est disponible au travers d'un moniteur d'exécution. Les questions se posent au moyen de requêtes de type SQL-Like relationnelles faites en langage "QUEL". Voici un premier exemple de ce langage :

Imaginons qu'un développeur veuille retrouver dans un programme des informations relatives aux appels de la méthode "buggy", par exemple, les valeurs des paramètres et le nom des appelants de cette méthode. Le problème est qu'au stade de la simple analyse du code source, les valeurs des paramètres ne peuvent pas encore être enregistrées sous forme relationnelle, c'est-à-dire qu'elles ne sont pas enregistrées dans la base de données de l'environnement. Pour nous permettre de formuler notre requête SQL-Like et d'obtenir le résultat escompté, Linton propose que le moniteur d'exécution se conforme également au modèle en représentant les valeurs dynamiques obtenues à l'exécution sous forme de relations. Dans le cas d'appels de méthode que nous analysons, c'est la relation suivante qui devra être évoquée :

```
callstack(procedure, level)
```

Ici, l'attribut "level" est la profondeur d'appel.
Utilisé dans une requête, cela donne ceci (Fig. 4.2) :

```
range of p is callstack
when p.procedure.name = "buggy" and p.level = max(p.level)
do
   print p
end
```

Fig. 4.2: Requête exprimée en langage "QUEL"

Maintenant si nous désirons exprimer une autre condition, par exemple n'afficher que les informations relatives aux appels de méthodes faites par la méthode "cause" il suffit de rajouter la ligne suivante (Fig. 4.3) :

```
range of p is callstack
range of q is callstack
when
   p.procedure.name = "buggy" and p.level = max(p.level) and
   q.procedure.name = "cause" and q.level = p.level - 1
do
   print p
end
```

Fig. 4.3: Requête exprimée en langage "QUEL"

Cet exemple illustre la manière dont Linton nous permet d'utiliser un langage relationnel pour questionner des éléments d'un code source. Une fine association entre modélisation et langage.

Avec la définition de ses 58 relations [22] , le potentiel de requêtes possibles semble très large, mais son système a été dénoncé comme étant très lent en terme d'exécution de requête [18].

Il comporte également une limite [20] [27], il ne supporte pas la récursivité, ce qui est plutôt ennuyant si nous voulons par exemple afficher les différents niveaux d'un graphe d'appels.

4.3 Prolog-Like

JQuery

JQuery [7] est un "Query Code Browser" développé au sein de la faculté universitaire d'informatique de British Columbia. C'est un outil qui s'intègre à Eclipse comme un plug-in au moyen duquel un utilisateur peut formuler ses questions à l'aide du langage logique TyRuBa.

Le principal avantage de cet outil repose sur son navigateur graphique qui permet une visualisation simultanée de plusieurs points de vue. Il permet, par exemple, de visualiser les hiérarchies d'héritage entre objets, tout en observant quelles sont les méthodes de ceux-ci ainsi que leurs appels. L'image en figure 4.4 illustre un tel résultat.

En ce qui concerne le TyRuBa, c'est un langage ressemblant très fortement à Prolog. Une différence visible est que les variables commencent avec le caractère ' ?' et non avec une majuscule. Cela permet de prendre le nom des méthodes, des classes et des variables comme des constantes sans devoir utiliser les guillemets.

Pour créer son vocabulaire, JQuery définit une série de prédicats spécifiques. En voici en figure 4.5 une liste non exhaustive.

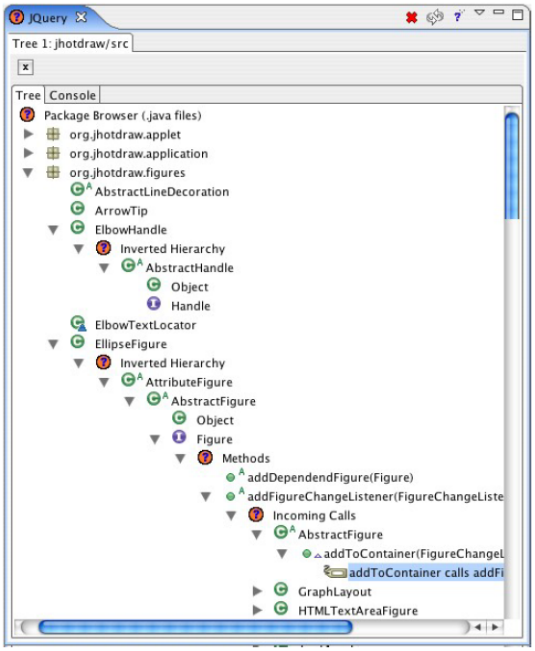


Fig. 4.4: Interface JQuery

Predicate	Description
package(?P)	?P is a package.
type(?T)	?T is a type defined in the program.
interface(?T)	?T is an interface defined in the program.
method(?M)	?M is a method defined in the program.
field(?F)	?F is a field defined in the program.
method(?T,?M)	?M is a method defined in type ?T.
returns(?M,?T)	Method ?M has return type ?T.
name(?E,?n)	Element (=package, type, method, field) ?E has name ?n.
re_name(?E,?regexp)	Element ?E has a name that matches ?regexp.
subtype(?Sub,?Sup)	?Sub is a direct supertype of ?Sup.
subtype+(?Sub,?Sup)	Transitive closure of subtype.
child(?E1,?E2)	?E2's declaration is directly nested inside ?E1.
reads(?reader,?field,?loc)	?field is read from ?reader at source location ?loc.

Fig. 4.5: Liste des prédicats définis dans Jquery

Ces prédicats permettent d'accéder à un grand nombre d'informations concernant la structure d'un programme étudié, les appels de méthode, le contexte de déclaration, la hiérarchie d'héritage, la localisation et les cibles d'un appel de méthode, les accès à un champ, où sont créés les objets, la localisation d'une erreur de compilation, les signatures de méthode, les tags, etc.

Il ne supporte pas la recherche de variable locale, la formulation de la généricité et la formulation des types d'expression [30].

Cela dit, nous relevons une approche intéressante pour exprimer la fermeture transitive d'une relation. Par exemple, pour la relation d'héritage entre classes, ils ont défini un nouveau prédicat

"subtype" en lui ajoutant un '+'.

Pour compléter ce mode simplifié, JQuery propose un mode avancé de configuration permettant de définir de nouveaux prédicats, mais cela requiert des connaissances plus approfondies du langage TyRuBa. Nous n'entrerons pas dans les détails de ce mode, mais selon [31] ce mécanisme de définition de nouveaux prédicats logiques sur base des autres prédicats existants, s'apparente plus à un mécanisme de type DSL-Like de type interne.

Recentrons-nous plutôt sur le système de requête simple en nous intéressant à un exemple concret [13] :

Imaginons un développeur qui voudrait corriger les problèmes de "naming convention" dans son programme. Il voudrait afficher toutes les classes héritant d'une même superclasse "Figure" et qui ne comporteraient pas le mot "Figure" dans leurs noms. Avec JQuery il peut composer la requête suivante (Fig. 4.6) :

```
name(?IFigure,Figure), subtype+(?IFigure,?Figure),
NOT( re_name(?Figure,/Figure$/) )
```

Fig. 4.6: Requête exprimée en langage TyRuBa

À ce niveau, le résultat peut sembler encore un peu difficile à exploiter, c'est pourquoi nous pouvons aisément décider de réorganiser les classes en les affichant dans leurs packages respectifs. Pour y arriver, JQuery permet de compléter la requête de cette manière (Fig. 4.7) :

```
selection:    name(?IFigure,Figure), subtype+(?IFigure,?Figure),
              NOT( re_name(?Figure,/Figure$/) ),
              package(?Figure,?P)
organization: ?P, ?Figure
```

Fig. 4.7: Requête exprimée en langage TyRuBa

SOUL

De son côté, la VUB (Vrije Universiteit Brussel) propose également un outil pour interroger du code source. Son langage "SOUL" [12] n'est pas aussi proche syntaxiquement de Prolog que l'est JQuery. Les concepteurs nous offrent un langage plus facile permettant une lecture plus naturelle.

Par exemple, ce qui se présenterait en Prolog comme :

```
isStatementIn(Inner,Outer)
```

s'écrit comme ceci avec SOUL :

```
if ?inner isStatementIn: ?outer
```

À noter que nous constaterons la même approche que celle de JQuery pour la définition des variables.

Cette volonté d'apporter un langage plus lisible semble en accord avec leur théorie [12]. Selon eux, il existe 2 facteurs qui font que les outils de "code querying" basés sur un langage logique ne connaissent pas encore de franc succès :

Le premier facteur est qu'il existe encore des cas où composer une requête logique n'est pas si facile. Par exemple, pour spécifier des caractéristiques d'un code source, la formulation logique peut très vite devenir complexe notamment pour l'expression de flux de contrôles et de données. De plus, les utilisateurs doivent être familiarisés avec des modèles établis, tels qu'une grammaire abstraite des noeuds d'un AST (abstract syntax tree).

Le second facteur concernerait la difficulté d'exploiter les résultats obtenus par d'autres applications.

Pour répondre à la première contrainte, Soul apporte des innovations avec une fonctionnalité permettant de reconnaître des modèles de code source.

Par exemple, le code suivant permet de retrouver un modèle de classes.

```
class Person {
  private Integer age;
  public Integer getAge() {  return this.age; }
  public Integer notGettingAge(Integer age) {  return age; }
}

if jtClassDeclaration(?c){
     class ?className {
       private ?fieldDeclarationType ?fieldName;
       ?modifierList ?returnType ?methodName(?parameterList) {
         return ?fieldName;
       }
     }
   }
```

Cependant, certaines constructions Java telles que les accès à un tableau, la généricité et les déclarations 'try/catch' ne sont pas encore supportées [30]. SOUL jouit également d'une conception basée sur Smalltalk depuis sa création. Comme nous pouvons l'observer dans la figure 4.8, il établit sa symbiose avec Java au travers d'une architecture particulière. Cette caractéristique permet la formulation d'expressions Smalltalk dans une requête. Syntaxiquement, cela se traduira par une expression entre crochets :

```
if ?m isMethodDeclaration,
   [?m getParent] equals: ?t,
   ?t typeDeclarationHasBodyDeclarations: ?l,
   ?i equals: [?l lastIndexOf_Object: ?m]
```

Cette requête signifie : "Quelle est la dernière méthode déclarée pour chacune des classes".

Malgré son architecture particulière et pour répondre au deuxième facteur précité, SOUL s'assure que sa représentation de l'AST soit l'AST d'Eclipse lui-même (Fig. 4.8). Ce qui permet aux autres applications voulant faire des modifications sur l'AST d'Eclipse, de ne pas récupérer des noeuds erronés lors d'une requête.

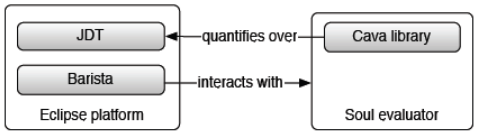


Fig. 4.8: Représentation logique de l'architecture de Soul

4.4 DataLog-Like

CodeQuest

CodeQuest a été introduit par l'université d'Oxford. Pour eux, l'importance de se baser sur un langage simplifié dont les limites de l'expressivité sont définies est capitale pour maintenir une bonne interactivité avec l'utilisateur.

CodeQuest propose une autre approche, utilisant Datalog comme langage de référence. Pour eux, le fait que DataLog soit un sous-ensemble de Prolog en fait un excellent candidat pour la réalisation d'outils de "code querying". Par exemple, il ne demande pas d'annotations logiques supplémentaires pour garantir la terminaison d'une requête. Il ne possède pas non plus de structures de données telles que les listes qui nous sont inutiles. Il supporte la récursivité, ce qui est intéressant par exemple pour les requêtes voulant exprimer la notion d'héritage. Ce langage aurait par essence un bon niveau d'expressivité pour accomplir le travail demandé.

En voici un exemple :

$$method(M), hasName(M, \text{'add'}), interface(L), hasName(L, \text{'List'}), hasChild(L, M)$$

Cette requête signifie que nous désirons toutes les interfaces 'List' qui contiennent une méthode appelée "add". Une remarque est tout de même à soulever concernant la formulation de la requête. Certains prédicats pourraient laisser croire à une utilisation facile du langage, ce qui n'est en réalité pas le cas. Par exemple si nous reprenons le cas précédent, un utilisateur consultant simplement la documentation des prédicats pourrait vouloir composer ceci :

$$hasChild(\text{'List'}, \text{'add'}).$$

Le système évaluera la requête comme 'false' car CodeQuest ne possède pas de système de "type" et le nom seul ne peut identifier un élément d'un programme.

Cela appuie à nouveau le fait que la composition de requêtes logiques n'est pas toujours triviale [12].

Tout comme les autres langages logiques, CodeQuest permet aussi l'extension du langage en utilisant la syntaxe suivante : Si, par exemple, nous voulions définir une nouvelle relation, nous pourrions écrire :

$$hasSubtype(T, S) :- extends(S, T) \ ; \ implements(S, T).$$

En résumé, DataLog est un langage plus compact que Prolog, ce qui facilite son apprentissage, mais CodeQuest ne nous apporte pas d'idées supplémentaires au niveau syntaxique pour interroger des éléments d'un code source. Nous avons souligné tout de même l'apparence trompeuse que peuvent avoir certains prédicats.

4.5 Autres approches

Un Système de requête sur base algébrique

Les explications qui suivent reposent sur un article publié en 1993 [27]. Cette parution est un peu vieille, mais elle propose de pallier aux manques de possibilités offertes par les systèmes de requête classiques de l'époque en terme de formulation de requêtes en introduisant un langage de type algébrique. Bien entendu, d'autres approches plus récentes ont été présentées dans la section précédente, mais les commentaires de cet article méritent d'être présentés.

Les auteurs avancent que les outils tels qu'OMÉGA, CIA et CIA++ [15] fournissent un système de requête formel, mais qu'ils restent trop faibles pour modéliser tous les détails structurels d'une application. D'autres modèles plus récents de l'époque tels que Rigi et Scan utilisent des graphes et des arbres syntaxiques abstraits (AST) pour raffiner l'information, mais ces outils restent handicapés par l'absence de langages procéduraux qui définissent bien les opérateurs. En conclusion, les chercheurs dénoncent le fait que la majorité des systèmes se retrouvent limités soit par leur modélisation soit par leur langage de requête.

Cet article avance une approche algébrique pour créer un système de requêtes qui répondrait à ces deux contraintes. En effet l'algèbre fournit une bonne base pour représenter l'information d'une part, mais également, elle possède déjà un ensemble d'opérations bien définies (+,-,*,...). La figure 4.9 définit plus précisément ces opérateurs .

Operator	Semantics
$+,-,*,/$	arithmetic operators
and,**or**,**not**	boolean operators
$=,<,>,\leq,\geq$	relational operators

Fig. 4.9: Opérateurs algébriques de base pour les types simples

Au niveau de la modélisation, cette approche propose de représenter le code source comme un ensemble d'objets encapsulant l'information concernant les composants, les relations entre les objets, les méthodes ainsi que toutes les autres informations pertinentes. Pour ce faire, le système définit une hiérarchie d'objets : par exemple l'objet "While-statement" est un sous-type de "Statement" qui est lui-même un sous-type de "Program-object". Ces objets peuvent être de types simples (CHAR, BOOLEAN, STRING,...) ou composés d'attributs qui sont eux-mêmes des objets (par exemple "While-statement" est composé d'une condition de type "Expression" et d'un corps de type "Statement"). La figure 4.10, en page 24, définit certains de ces objets.

Pour compléter les types de base, une série d'opérateurs supplémentaires ont été définis et sont repris en figure 4.11. Ils nous permettent de faire des constructions de requêtes. Pour illustrer notre propos, voyons ce que donne la formulation d'une requête :

Imaginons que nous voulions retrouver toutes les fonctions définies dans un fichier analyser.c, il suffit d'écrire :

$$retrieve_{funcs}(select_{name=analyzer.c}(FILE))$$

De manière plus complexe maintenant, imaginons que nous voulions obtenir le fichier qui possède le plus de fonctions :

$$head_1(order_{no\_of\_func.>}(set\_to\_seq(extend_{no\_of\_func:=size\_of(funcs)}(FILE))))$$

Avec ce langage de "bas niveau", les possibilités offertes sont très larges. Il est facile d'imaginer la création d'un langage de plus haut niveau par dessus.

```
type    DECLARATION-LIST            set of DECLARATION
type    STATEMENT-LIST              sequence of STATEMENT
....
type    COMPOUND-STMT               subtype of STATEMENT
                                    ......
                                    decls:DECLARATION-LIST (composition)
                                    stmts:STATEMENT-LIST (composition)
endtype
type    FUNC-CALL                   subtype of EXPRESSION
                                    ......
                                    funcdef:FUNCTION (reference)
                                    arguments:EXPR-LIST (composition)
endtype

type    FUNCTION                    subtype of PROGRAM-OBJECT
                                    ......
                                    type-spec:TYPENAME (composition)
                                    name:STRING (composition)
                                    parameters:PARAM-LIST (composition)
                                    body:COMPOUND-STMT (composition)
endtype

type    FILE                        subtype of PROGRAM-OBJECT
                                    ......
                                    name:STRING (composition)
                                    funcs:FUNCTION-LIST (composition)
                                    globdecls:DECLARATION-LIST (composition)
endtype

type    STATEMENT                   subtype of PROGRAM-OBJECT
                                    ......
                                    line-no:INTEGER (annotation)
                                    uses:VARIABLE-LIST (reference) inverse used-by
                                    defines:VARIABLE-LIST (reference) inverse defined-by
                                    live:VARIABLE-LIST (method) live-compute
endtype
.....
```

Fig. 4.10: Objets permettant de représenter le code source

Operators	Description
$< attribute >$	*signature*: COMP ⟶ ANY *syntax*: $< attribute > (< object >)$ *semantics*: Returns the value of the specified attribute
select	*signature*: COLLECTION(ANY1) ⟶ COLLECTION(ANY1) *syntax*: $select_{<boolean\ expression>}(< objectcollection >)$ *semantics*: Chooses a subcollection based on a condition
project	*signature*: COLLECTION(COMP1) ⟶ COLLECTION(COMP2) *syntax*: $project_{<attributelist>}(< objectcollection >)$ *semantics*: Retains only the specified attributes
extend	*signature*: COLLECTION(COMP1) ⟶ COLLECTION(COMP2) *syntax*: $extend_{<attribute:=algebraic\ expression>}(< objectcollection >)$ *semantics*: Adds a new attribute to each object
retrieve	*signature*: COLLECTION(COMP) ⟶ COLLECTION(ANY) *syntax*: $retrieve_{<attribute>}(< objectcollection >)$ *semantics*: Retrieves a specified attribute
closure	*signature*: COMP ⟶ SET(COMP) *syntax*: $closure_{<attributelist>}(< object >)$ *semantics*: Finds all objects reachable using listed attributes
apply	*signature*: COLLECTION(ANY1) ⟶ COLLECTION(ANY2) *syntax*: $apply_{<operator>}(< objectcollection >)$ *semantics*: Applies a unary operator to each element

Fig. 4.11: Opérateurs pour manipuler les Objets et Collections

Langage de base de données en graphe

Pour Raoul-Gabriel Urma et Alan Mycroft [31], la majorité des outils d'interrogation de code source utilisent ou utilisaient des bases de données relationnelles ou déductives principalement à cause de la limitation de la mémoire des ordinateurs de l'époque. Mais ce choix a comme conséquence de ne fournir qu'un sous-ensemble de l'information contenue dans un code source.

Ils affirment également que les autres représentations du code source plus complètes telles qu'un simple String, un arbre syntaxique et un graphe de contrôle de flux sont très optimisées seulement pour certains types de requêtes, mais ne le sont pas spécialement pour d'autres. Par exemple, un simple string est suffisant pour les recherches textuelles ou lexicales. Pour des requêtes sur la structure du programme, un arbre syntaxique est plus efficace et pour les questions sur les flux de contrôle, un graphe de contrôle de flux est privilégié.

Un des problèmes des systèmes n'utilisant qu'une seule représentation est que les utilisateurs sont dès lors limités à ne pouvoir exprimer qu'un seul type de requête alors que la majorité des questions exprimées dans le cadre de la compréhension de programme sont mixtes, par exemple "Trouver une variable qui contient le mot 'point' dans son nom dont le type est une classe ou une sous-classe de 'Element' et qui est consulté dans une méthode appelée récursivement".

Pour R-G. Urma et A. Mycroft, une base de données en graphe permet d'unifier toutes les représentations au sein d'un seul modèle. Dans leur présentation [31],ils utilisent la base de données Neo4j pour mettre en place un tel prototype de modèle unifié.

Le langage pour interroger la base de données n'est pas spécifique à l'interrogation de code source et peut donc, comme le montrent les requêtes suivantes, être un peu lourd en terme d'expressivité notamment avec l'ajout de termes à préoccupation technique pour l'indexation (voir Fig. 4.12).

```
START m=node:node_auto_index(nodeType='ClassType')
MATCH path=n-[r:IS_SUBTYPE_EXTENDS*]->m
WHERE m.fullyQualifiedName='java.lang.Exception'
RETURN path;
```

```
START m=node:node_auto_index(nodeType='JCMethodDecl')
MATCH c-[:DECLARES_METHOD]->m-[:CALLS]->m
RETURN c,m;
```

Fig. 4.12: Requêtes faites avec le langage 'Cypher' de Neo4J

Mais nous avons relevé deux expressions très intéressantes :

1) La possibilité de former des modèles relationnels avec des flèches '-[]->' pour exprimer des successions d'appel ou des relations d'héritage.

2) La possibilité d'exprimer sur une relation une fermeture transitive en ajoutant le caractère '*'

Systèmes de requête en langage naturel

Une des approches les plus récentes en matière de systèmes basés sur un langage d'interrogation naturel est Evolizer [20] dont nous pouvons observer la démarche en figure 4.13

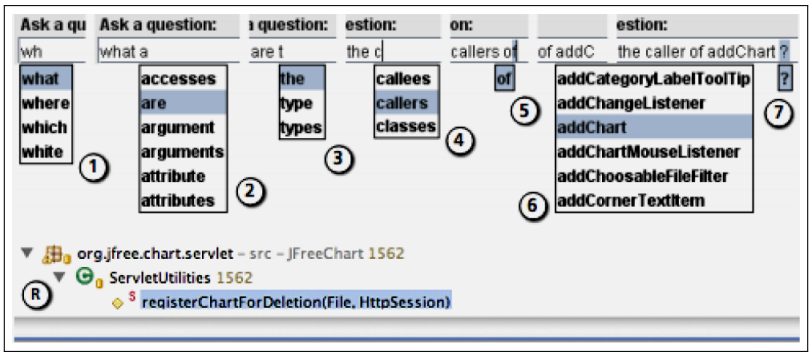


Fig. 4.13: Formulation de requêtes faites sous Evolizer

Dans ce système, la composition des requêtes est limitée par un choix restreint de possibilités. Pour créer son catalogue de requêtes, ce système s'est inspiré du système Ferret de De Alwis et Murphy [11], qui propose 36 requêtes permettant de répondre à la majorité des questions qu'un développeur pourrait se poser face à du code. Elles concernent principalement des questionnements liés à la structure du code, aux informations structurelles statiques et aux informations sur les flux de contrôle, telles que les relations interclasses, intraclasses, d'héritages, de déclarations, etc.

Une seconde approche [21] propose un système de requêtes en langage naturel débridé n'imposant pas que la composition d'une requête se fasse au travers de listes restreintes de mots tels que le fait le système Evolizer dans son interface. Elle demande simplement que la question exprimée soit grammaticalement correcte.

Cette approche plus ouverte soulève un certain nombre de problèmes quant à la compréhension exacte de ce qui est demandé au système. Celui-ci travaille avec une méthode de l'outil JDT d'Eclipse permettant d'interroger le code source. Afin de fournir les bons paramètres à la méthode, les concepteurs ont mis en place un processus d'analyse de la question passant par plusieurs étapes qui sont représentées dans la figure 4.14. Leur processus d'extraction repose sur des techniques telles que "Part of speech taging" et "stemming".

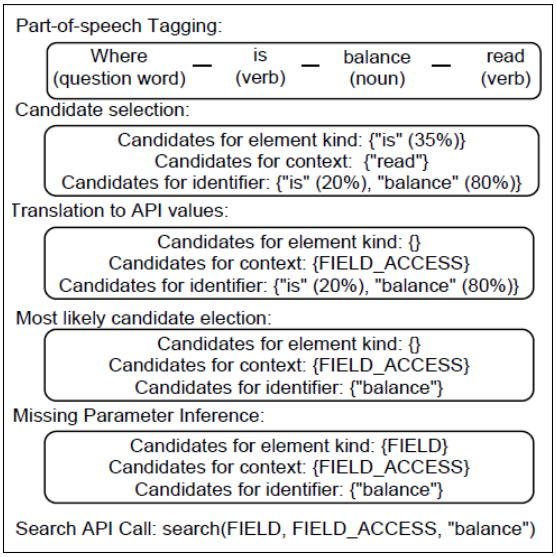


Fig. 4.14: Formulation de requêtes faites sous Evolizer

4.6 Conclusion

Nous avons vu un ensemble d'outils intéressants dans le domaine de l'interrogation de code source. La majorité d'entre eux se concentre seulement sur une analyse statique du programme à l'aide d'une représentation du code (modèle relationnel, AST...). Mais il en existe d'autres, comme OMEGA, qui fournissent également des mécanismes afin de récupérer les valeurs dynamiques issues de l'exécution du programme.

Parmi les langages utilisés pour formuler des requêtes, nous avons pu observer plusieurs approches différentes. Allant de l'emploi de langages structurés (relationnel, logique, graphe) à l'expression de questions en langage naturel.

Avec l'analyse d'un langage "SQL-like de type relationnel", nous avons relevé l'importance d'apporter une attention particulière à la formulation de requêtes récursives, et ce, afin d'exprimer des arbres d'appel ou des relations d'héritage entre classes.

Avec l'utilisation d'un langage logique tel que le propose Prolog ou Datalog, la formulation de requêtes récursives est possible. Cependant, la difficulté d'exprimer des requêtes pour analyser plus en détail une hiérarchie de classe ou un flux de donnée reste omniprésente [30].
Nous avons relevé un avantage à l'utilisation de ce type de langage comme base pour offrir deux types d'utilisations aux outils, une première simplifiée ne demandant pas une grande connaissance de la programmation logique (les prédicats sont déjà définis) et une seconde utilisation, plus complexe, permettant la définition de nouveaux prédicats, ouvrant ainsi la porte à une extension des possibilités de requêtes.
D'un point de vue syntaxique, nous avons observé des initiatives intéressantes pour faire évoluer la syntaxe de Prolog pour permettre une lecture plus naturelle des requêtes d'une part et pour permettre l'expression de modèle de classe d'autre part.

Un langage se basant sur l'algèbre a été avancé et il constitue principalement une formulation d'assez "bas niveau" offrant une nouvelle couche assez flexible pour la définition d'autres langages par dessus. Cette démarche a permis principalement de prouver la faisabilité d'un tel langage. Cela dit, nous n'avons pas remarqué qu'il apportait de grandes nouveautés en terme de simplicité d'expression.

Un prototype utilisant une base de données en graphe a été présenté, soulevant l'importance que peut avoir la modélisation d'un code source sur les capacités d'un système à exprimer des requêtes. Du point de vue du langage de requête de base de données en graphe, cela nous a permis de découvrir une manière très élégante d'exprimer, grâce à des flèches, des modèles d'appel et des relations d'héritage.

Deux outils basés sur des langages naturels ont également été avancés. Nous avons commencé par une approche plus balisée proposant une formulation de requêtes par l'utilisation de listes de mots prédéfinis, puis nous avons continué avec l'explication d'une seconde, plus ouverte et ne demandant que le respect de la grammaire comme seule contrainte. Nous constaterons cependant que dans le cas de la formulation de requêtes non balisées, il existe un réel problème d'interprétation de la requête, ouvrant la porte à des erreurs de compréhension par le système (83 % de

réussite [21]). Nous pouvons rajouter que cet effet indésirable n'en serait que plus accentué par l'arrivée de nouvelles fonctionnalités, par exemple, pour la détection de modèles de classe comme nous avons pu le voir dans SOUL. Dans ce dernier cas, il serait plus difficile encore de formuler une telle requête de manière naturelle.

5. L'ANALYSE DYNAMIQUE

Maintenant que nous avons fait un tour d'horizon des différents types de langages utilisés pour exprimer une interrogation en rapport avec du code source, intéressons-nous d'un peu plus près à l'analyse dynamique par la récolte de traces d'exécution et voyons ensemble de quelle manière nous pourrions mettre en place un système de requêtes qui répondrait au mieux à nos besoins.

5.1 Traces d'exécution

Des traces d'exécution sont des informations récoltées durant l'exécution d'un programme. Grâce à elles, il est possible de découvrir des informations sur ce qui s'est passé dans l'univers caché d'une application durant son fonctionnement. Par exemple, une trace peut nous indiquer qu'une méthode a été appelée, qu'une classe a été chargée en mémoire, ou encore nous donner la valeur d'un argument passé en paramètre à une méthode.

Dans cet article [25], les auteurs définissent les traces d'exécution de la sorte : "Des traces d'exécution (également connues sous le nom de "traces de programme") sont des descriptions d'événements qui se produisent durant l'exécution d'une application logicielle."

L'exploitation de traces d'exécution nous amène à nous intéresser à deux matières [24]. La première concerne la phase d'acquisition durant laquelle sont interceptées et enregistrées les informations composant la trace. La seconde concerne la phase d'analyse dans laquelle les traces sont soumises à différents traitements.

Pour bien comprendre la différence de fonctionnement entre analyses statique et dynamique telle que nous la présentons, voici en figure 5.1 une représentation succincte des processus d'utilisation des 2 approches.

Selon Tamar Richner et Stéphane Ducasse de l'université de Bern [28], l'approche dynamique (Fig. 5.1) est très attirante pour comprendre un programme par rapport à l'analyse statique pour les quelques raisons suivantes :

1) La récolte d'informations se fait durant l'exécution d'un scénario, ce qui permet de réduire considérablement la portée de l'investigation.

2) L'information dynamique est toujours précise par rapport au scénario exécuté, elle est fiable par rapport à ce scénario.

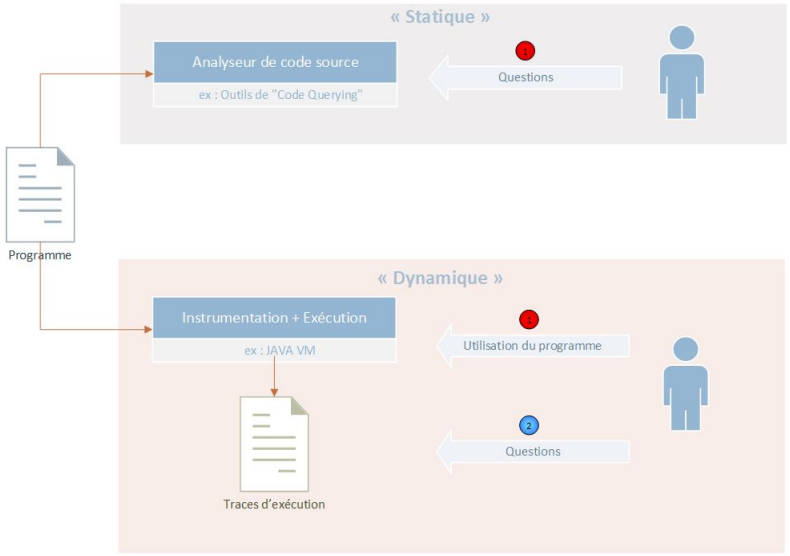


Fig. 5.1: Processus d'utilisation

3) Obtenir des traces d'exécution est un procédé relativement simple pour analyser des flux de contrôle.

4) Une trace d'exécution peut fournir des informations qui ne pourront jamais être obtenues grâce à l'analyse statique, telle que le nombre d'instances ou la multiplicité des relations entre objets. Nous verrons d'autres cas plus précis dans la section réservée aux besoins et cas d'utilisation de traces d'exécution.

Malgré ces avantages, il est important de relever quelques points délicats :

1) L'analyse dynamique implique de récolter, d'enregistrer une grande quantité d'informations [25] et de parfois même les manipuler directement. Cela engendre un ralentissement et une perte de performance sur le programme observé. Certaines approches tentent de réduire cet effet [24] en ne prenant que les informations (événements) pertinentes et en procédant à l'analyse des données après l'exécution.

2) Une deuxième remarque à souligner [26] à propos de l'analyse dynamique est qu'il n'est pas toujours facile d'obtenir une vision globale de l'application, cela implique en effet d'utiliser l'entièreté des fonctionnalités.

C'est pour cette raison notamment que l'analyse statique et dynamique sont généralement vues comme complémentaires [26] [28].

5.2 Récolte de traces d'exécution en Java

Dans le cadre de ce mémoire, nous nous concentrerons principalement sur la récolte de traces d'exécution de programmes Java. Pour récolter des traces d'exécution, nous sommes généralement amené à injecter des lignes de code supplémentaires dans le code source du programme, notamment pour faire appel à des méthodes de traçage et enregistrer les informations intéressantes.

Il existe plusieurs approches pour arriver à ce résultat :

5.2.1 L'instrumentation statique

L'instrumentation statique permet de modifier directement le code source ou le bytecode (.class) d'un programme. Dans le cadre d'une modification de code source, on commence généralement par une analyse lexicale et syntaxique afin de placer des sondes dans le code. Ainsi il est par exemple possible de découvrir des structures de contrôle de flot et d'en déduire d'éventuelles branches mortes qui y sont reliés. Dans le cas où nous ne disposons pas du code source, il est quand même possible d'injecter du bytecode supplémentaire grâce à des bibliothèques, comme BCEL (Apache Jakarta), SERP ou plus récemment ASM. Par exemple AspectJ, qui se base sur la technique de la programmation orientée Aspect, permet un tissage statique sur le bytecode en utilisant la bibliothèque ASM.

5.2.2 L'instrumentation dynamique

Il existe 2 principales techniques. Une des ces techniques consiste à intercepter les événements intéressants au sein de la machine virtuelle grâce à des hooks, mais elle est relativement compliquée à mettre en œuvre [26]. La seconde, plus simple, consiste à instrumenter le bytecode des classes Java à leur chargement.

Depuis sa version 1.5, Java fournit un mécanisme très efficace pour accéder au bytecode au moment où il est chargé par la JVM. Grâce à l'introduction d'Agents au sein de la JVM, il est possible de modifier du code déjà compilé sans pour autant avoir des connaissances de trop bas niveau. Manipuler du bytecode directement reste une tâche complexe surtout si l'on ne s'y connaît pas. C'est pour cela qu'avec cette méthode il est possible de charger des outils qui permettent de faciliter cette tâche. Par exemple Javassist est un des frameworks les plus connus qui permet de s'abstraire du bytecode en fournissant des méthodes de plus haut niveau.

Sans trop sortir du cadre de ce mémoire, mais dans un souci de ne pas laisser le lecteur sur sa faim en terme d'explication, voici comment mettre en place une instrumentation grâce à cette technique :

La JVM fournit la possibilité de charger avec son ClassLoader un agent au démarrage d'une application. Pour associer un agent, il suffit de lui rajouter l'option -javaagent :jarpath[=options] en ligne de commande. Cet agent se présente sous la forme d'un ". jar" devant comprendre une implémentation de la méthode :

"public static void premain(String agentArgs, Instrumentation inst)"

Dans cette méthode, il est possible d'instancier et d'enregistrer une implémentation d'une interface qui nous intéresse, "ClassFileTransformer" . C'est en effet dans cette classe que nous pouvons manipuler le byteCode et que nous sommes amenés à injecter du code supplémentaire. Pour être plus précis, c'est la méthode "public byte[] transform(...)" qui réalise cette tâche notamment grâce aux méthodes de Javassist.

5.2.3 Programmation orientée Aspect

Le paradigme orienté Aspect est très apprécié pour instrumenter du code. Il peut se faire à différents moments [14] : statiquement, avant ou après la compilation du bytecode, ou dynamiquement à l'exécution. Les implémentations fournissent généralement un langage spécifique très adapté et flexible pour réaliser une instrumentation à des endroits précis d'un programme. Par exemple, avec outils AspectJ pour Java, nous pouvons définir ce qu'on appelle un "pointcut" qui cible dans un code source un ensemble de "joinpoints" autour desquels sera injecté le code supplémentaire. Plus concrètement, nous pourrions définir un "pointcut" (ensemble de "joinpoints") qui représente tous les appels de méthodes d'un programme et y attacher du code qui enregistrerait des informations relatives à ces appels de méthodes.

5.3 Données récoltées

Comme expliqué plus haut, une trace d'exécution est la description d'un événement survenu durant l'exécution d'un programme.

Voici une liste de ces différents événements enregistrables par instrumentation de code : [25].
1) création d'objet
2) appel de méthode
3) retour d'appel de méthode
4) accès à un champ
5) modification d'un champ
6) lancement et manipulation d'exception
7) Chargement d'une classe

D'un événement à l'autre, le nombre et le type d'informations à relever sont différents. Typiquement, une trace d'exécution est composée d'une indication sur le type de l'événement, le moment où il s'est produit ("estampille" [24]) et d'un ensemble d'informations complémentaires.

Voici en figure 5.2, un exemple de deux traces d'exécutions enregistrées dans un fichier XML. Bien entendu, chaque système aura sa propre manière d'enregistrer ses traces. Mais celle-ci nous a paru adéquate comme première illustration pour le lecteur :

Nous y distinguons clairement les balises représentant un événement "création d'objet" suivi d'un autre "appel de méthode".

Dans le premier événement, nous avons l'information sur le type de l'objet créé ("TestClass1") et son identifiant ("object1"), et dans le second événement qui est un appel de méthode, nous avons l'identifiant de l'objet appelant("object2"), l'identifiant de l'objet appelé("object1"), le nom de classe de l'objet appelé("TestClass1"), le type du paramètre passé à la méthode(String) et son identifiant("object4").

Généralement, l'information des traces est enregistrée dans un fichier commun et représentée par une structure de données plates à plusieurs champs.

Afin d'avoir une vue plus globale et générique des informations récoltables, l'institut de technologie de Géorgie a dressé une liste théorique de ce qu'ils appellent des "entités d'instrumentation" utiles pour une analyse dynamique [9] (voir figure 5.3). Pour chaque entité, ils ont associé des informations accessibles par la POA (Programmation Orientée Aspect). Dans ce cas, ce concept d'entité d'instrumentation est comparé au principe de "pointcut" proposé par "AspectJ [2]". Par

```
<objectcreation objectid="object1">
  <complextype>
    <typename>
      test_classes.TestClass1
    </typename>
  </complextype>
</objectcreation>
<methodcall receiverid="object1"
            senderid="object2">
  <methodname>method2</methodname>
  <typename>
    test_classes.TestClass1
  </typename>
  <parameter>
    <objectvalue objectid="object4"/>
    <complextype>
      <typename>
        java.lang.String
      </typename>
    </complextype>
  </parameter>
</methodcall>
```

Fig. 5.2: Traces d'exécution représentant les événements : Une création d'objet et un appel de méthode

cette représentation, il appuie le fait qu'il est possible de rendre flexible et générique la phase d'instrumentation.

Instrumentable entity	*Information available*
Method entry	enclosing object argument objects
Method exit	return object or exception object
Before method call	target object parameter objects
After method return	return object or exception object
Field read	field object containing object
Field write	old field object new field object containing object
Start of basic block	*none*
End of basic block	*none*
Before a branch	*none*
After a branch	*none*
Throw	exception object
Catch	exception object
Predicate	evaluated predicate result
Acyclic path	*none*

Fig. 5.3: Liste d'informations triées par "entité d'instrumentation" [9]

5.4 Exploitation des traces

Cette section nous offre quelques cas d'utilisation plus spécifiques à l'analyse dynamique. Ceci dans le but de fournir un ensemble supplémentaire d'opérateurs à utiliser dans notre langage. D'autres cas seront rencontrés durant la composition du langage.

Comme expliqué par V.Marangozova et G.Pagano [24] (section sur l'exploitation de traces), la majorité des solutions existantes travaille sur une représentation visuelle de l'information et non sur l'extraction de connaissances d'un niveau sémantique plus élevé.

Pour nos interrogations, nous avons retenu les cas d'utilisation suivants :

Calcul du temps d'exécution

Le calcul du temps d'exécution est une opération relativement simple à mettre en œuvre. Comme expliquée dans la présentation des données, une trace comprend souvent une information sur le moment où s'est produit l'événement. Si nous prenons la trace qui se réfère au moment où on est entré dans une méthode et la trace qui informe sur la sortie de la méthode, on peut facilement poser une question sur le temps d'exécution. On peut également étendre ce principe à d'autres segments du scénario d'exécution, du moins tant que nous possédons deux valeurs de temps.

Un point doit, cela dit, être soulevé : le temps d'exécution seul n'est pas toujours intéressant, par contre, comparer différents temps d'exécution peut l'être.

Calcul statistique

Le calcul statistique [24] ouvre également une voie dans le domaine de la comparaison de résultats. Par exemple, il peut être intéressant de voir combien de fois une méthode a été appelée durant l'exécution d'un scénario, de comparer un résultat par rapport à une moyenne.

Calcul de distance d'appel

Une autre valeur intéressante à calculer : la "distance d'appel" [26]. Il est effectivement possible avec un agent JVM d'enregistrer les "stacktraces" et d'en déduire la distance d'appel. Nous pourrions ainsi poser des questions afin de relever tous les appels de méthodes inférieures, supérieures ou égales à une distance d'appel.

Détection de design pattern

En poussant la réflexion plus loin, avec la distance d'appel associée au "stacktrace", nous pourrions chercher à vouloir détecter l'utilisation de design patterns dans un programme [26]. Bien entendu, les programmes étant très variés, le travail d'identification peut être complexe. L'analyse dynamique peut certainement présenter sur ce point un avantage. En effet, comme expliqué précédemment, un des avantages de l'approche dynamique par rapport à l'approche statique est que l'analyse se fait suite à l'exécution d'un scénario, ce qui permet de réduire considérablement la zone d'investigation.

Analyse de contextes particuliers

Un cas intéressant à aborder concerne les appels de méthode dans certains contextes particuliers de "disign patterns", par exemple, dans le cas du design pattern "Publish-Subscribe" : Il est ici difficile de détecter statiquement quand une méthode est appelée ou quand elle ne l'est pas. L'analyse dynamique est particulièrement indiquée dans ce genre de cas. Elle permet effectivement d'enregistrer une trace des appels à cette méthode mais également une partie de son contexte d'exécution.

Calcul d'état

Poser des questions sur le changement d'état d'un programme peut également être utile [24]. Malheureusement, c'est une information métier qui ne se retrouve généralement pas dans les traces d'événement en temps que tel. Il est cela dit possible de le déduire dans certains cas par un traitement particulier. Par exemple dans les applications MPI (Message Passing Interface), il est possible de déduire un état de communication entre les processus en retrouvant un événement d'envoi de message et un événement de réception de message.

Détection de motifs

La détection de motifs [24] consiste à trouver des endroits où l'exécution d'un programme est perturbée ou erronée. Par exemple, nous pourrions vouloir poser des questions du genre : "Donne-moi les endroits où l'exécution se passe comme prévu et/ou ceux où il y a un problème". Dans le cas des applications MPI, nous pouvons détecter des états de communication et donc cela peut nous permettre de demander celles qui connaissent un état de communication ralenti ou celles qui connaissent un état de communication normal.

6. COMPOSITION D'UN LANGAGE

Seront présentés dans ce chapitre, les fonctionnalités choisies pour la réalisation de notre langage, nos justifications et nos choix en terme d'implémentation suivit d'une longue partie présentant les bases de notre langage. Pour conclure ce chapitre, nous présenterons un scénario de test pour mettre à l'épreuve nos choix théoriques et nous tirerons des conclusions sur ce scénario.

6.1 Fonctionnalités de base

Filtrer [24] [25]

Un des besoins qui ressort le plus souvent de la littérature est celui de filtrer les événements. Que ce soit dans un but de simples consultations ou de représentations graphiques, il faut impérativement pouvoir filtrer l'information. Il est souvent recommandé de le faire au niveau de la phase d'instrumentation pour des raisons de performance, mais le filtrage au moment de la phase d'exploitation reste nécessaire. Un langage de requêtes doit impérativement permettre de réaliser cette tâche.

Agréger [24]

Un problème souvent rencontré dans le domaine de la visualisation d'événements est l'encombrement et la surcharge de l'information présentée. La solution généralement présentée pour résoudre ce problème est l'utilisation de fonctions d'agrégation sur les traces afin d'en diminuer le nombre à considérer. Par exemple, une somme est certainement la fonction d'agrégation la plus utilisée. Au lieu de présenter de manière détaillée un ensemble de résultats dans un tableau, une somme nous fournit le nombre de résultats obtenus.

Grouper [24]

Un regroupement est souvent employé de pair avec une fonction d'agrégation. Comme son nom l'indique, il permet de regrouper une information par catégorie ou par type. Par exemple, il est courant de voir associée à une trace d'exécution une information sur le 'thread' responsable de son exécution. Dès lors, il peut être intéressant d'afficher l'information regroupée par processus d'exécution.

Composer

Le caractère compositionnel d'un langage de requête est une caractéristique intéressante, il permet de composer une requête à partir de plusieurs autres. Nous pouvons retrouver cette caractéristique à plusieurs niveaux. Soit en utilisant le résultat d'une requête lors de la comparaison

d'une valeur, notamment avec l'utilisation d'une fonction d'agrégation, soit en utilisant une requête pour créer un sous-ensemble de recherche supplémentaire. Par exemple, pour composer des requêtes du style, "Trouve-moi dans cet ensemble-ci tout ce qui ne se trouve pas dans ce sous-ensemble-là".

Abstraire

Une dernière caractéristique intéressante que peut avoir un langage est la capacité à nommer des constructions de la requête afin qu'elles puissent être référencées dans d'autres contextes ou d'autres parties de la requête.

6.2 Choix et justifications

En plus de répondre correctement aux fonctionnalités de base présentées précédemment, nous voudrions un langage capable d'exprimer visuellement une suite d'événements tels qu'ils se sont produits durant un flux d'exécution d'un programme.

Une manière efficace de représenter un flux d'exécution est un arbre d'appel. L'approche la plus intéressante que nous avons vue pour la réalisation de ce type de requête est certainement le prototype de R-G. Urma et A. Mycroft qui permet l'expression de modèles en graphes. Mais le langage reste intimement lié à la base de données Neo4J [3] et n'est dès lors pas spécifique à l'interrogation de code source.

Il présente une verbosité technique d'une part, et d'autre part, il impose d'exprimer chaque concept de programmation avec une flèche directionnelle ce qui peut être perturbant dans certains cas. Par exemple, un lien d'appartenance entre une classe et une méthode s'exprime avec une flèche directionnelle au même titre que pour un appel de méthode. D'un point de vue dynamique, un appel de méthode implique le déplacement du processus d'exécution d'un point vers un autre alors qu'une relation d'appartenance entre une classe et une méthode ne le sous-entend aucunement. Nous devons pouvoir exprimer de manière plus claire certains concepts non directionnels.

Différentiation visuelle d'opérateur

L'importance de différencier visuellement des opérateurs en fonction de leurs caractéristiques communes se justifie par le besoin des utilisateurs à vouloir regrouper mentalement les opérateurs exprimant des concepts communs. Ce principe de mémorisation est d'ailleurs pris en compte dans la réalisation d'interfaces graphiques de programmes, par exemple, les boutons servant à fermer une fenêtre vont avoir tendance à se présenter de la même manière.

Dans son livre "First Principles of interaction Design" [29], B. Tognazzini fait référence à la cohérence, l'apprenabilité et l'utilisation de métaphores pour améliorer l'interaction avec l'utilisateur. Par son principe de cohérence, B. Tognazzini nous dit : "utiliser les mêmes conventions pour les mêmes choses, sinon les différencier".

Si nous appliquons ces principes à un langage, en plus d'éliminer les préoccupations techniques liées à une base de données, nous pouvons apporter une véritable amélioration à la composition et à la lecture d'une requête.

Par exemple, dans notre cas, nous pourrions représenter toutes relations structurelles d'un programme par un point (Eg. Package.Class.Methode.Field), tous les déplacements dynamiques faisant référence à un événement bougeant le flux d'exécution d'un point A à un point B par une flèche simple "–>" (Eg. appel de méthode) et tous les événements ne bougeant pas le flux d'exécution vers un autre élément par une double flèche sans barre"»" (Eg. lecture d'un champ).

De la sorte, nous pouvons représenter un modèle d'exécution comme ceci :

```
c.m1 --> m1 --> m2 >> x
```

Fig. 6.1: Requête décrivant un appel potentiellement récursif suivi d'un appel vers une méthode m2 qui accède à un champ x

Dans cet exemple, nous comprenons que le flux d'exécution se déplace d'un élément 'm1' possédant un parent 'c' vers un autre élément 'm1' et que cela pourrait représenter une potentielle forme de récursion, et puis, que le flux se déplace vers un élément 'm2' où va se produire un événement sur un élément 'x' sans pour autant déplacer le flux d'exécution vers ce dernier.

Ce premier résultat, est un peu minimaliste et bien que certaines choses peuvent ressortir naturellement d'autre reste à définir. Par exemple, nous n'avons pas encore précisé si le parent 'c' du premier élément 'm1' est implicitement le parent du second élément 'm1' et il ne précise pas non plus comment le flux d'exécution s'est retrouvé sur l'élément de départ 'c.m1'.

Même si ce test présente à ce stade encore certaines lacunes, il nous a montré qu'il est possible d'aller dans ce sens tout en gardant une bonne compréhension du langage.

A titre de comparaison, les figures 6.2 et 6.3 nous rappellent ce que proposait respectivement le langage 'Soul' et le langage 'Cypher' pour définir un lien de parenté entre une méthode et son parent .

```
[?m getParent] equals: ?t,
```

Fig. 6.2: Langage Soul : Lien de parenté

```
c-[:DECLARES_METHOD]->m
```

Fig. 6.3: Langage Cypher : Lien de parenté

Cela dit, comme une solution à un problème en entraîne souvent un autre, nous verrons que le défi de notre DSL reste entier : en voulant ajouter l'aspect dynamique d'un programme au sein d'une requête, nous ajoutons une complexité visuelle importante qui nous éloigne indéniablement de ce premier exemple.

En effet, les questions des développeurs ne se limitent généralement pas à l'expression d'un type d'information, mais concernent plutôt l'expression de requêtes mixtes impliquant plusieurs niveaux d'information [31]. L'analyse dynamique vient ajouter sa part de concept et d'informations. Tel que le concept d'instance, de valeur d'argument et de temps d'exécution.

Choix des symboles

Il faut savoir que durant notre réalisation, un grand nombre de petits essais visuels ont été entrepris. Nous ne voulons pas rentrer dans les détails rébarbatifs de chaque décision impliquant le choix d'un caractère plutôt qu'un autre. Voici typiquement, en figure 6.4, à quoi ressemble un test visuel qui nous a permis d'arriver à notre version du langage. Le modèle de requête de ce test n'a pas comme but d'être compris, mais plutôt de faire ressortir les incompatibilités visuelles entre caractères. Dans ce test, chaque ligne est identique aux autres, mais représente une version différente du langage. Nous avons ensuite comparé visuellement ces lignes entre elles et nous avons procédé à une élimination des candidats les moins compréhensibles. Nous considérons que ce test est suffisant à notre niveau de prototypage.

Fig. 6.4: Test visuel sans importance sémantique

6.3 Bases du langage

À ce stade, il ne s'agit que de "choix théoriques". Nous tirerons des conclusions sur les problèmes et limites de notre approche avec une mise à l'essai de notre langage.

Dans la suite de ce mémoire, nous appellerons entité (ou élément) de programme, les 'packages', les classes/interfaces, les blocs d'initialisation, les méthodes, les constructeurs, et les champs.

Sur base des différents événements que nous avons listés dans le chapitre traitant de l'analyse dynamique (section 'Données récoltées'), nous avons créé en figure 6.5 une liste d'expressions qui nous serviront à créer des modèles de flux d'exécution, tels que nous les avons imaginés dans la section précédente. Chaque expression est composée d'une flèche qui représente l'aspect dyna-

mique de l'événement et d'une entité de programme (lettre 'B') sur laquelle nous travaillerons une description plus statique (nous y reviendrons plus tard). A ce stade de l'explication, le fait qu'une entité de programme soit représentée par 'B','X',... ou 'Z' n'a pas d'importance. Il faut juste retenir que 'B' est une variable(identificateur de variable) et qu'elle pourra être référencée à d'autres endroits de la requête.

Nous avons aussi choisi d'exprimer en majuscule les variables composant nos expressions afin d'accentuer l'effet visuel. Dans l'étude du 'code querying', certains langages de type Prolog-Like utilisaient le point d'interrogation comme solution, mais, une fois testé, il s'est avéré inadapté à notre type de langage parce qu'il rend la lecture plus difficile. Ils ne faut pas les confondre avec les symboles 'c','m','b','t','r','w','x','/x\' qui sont réservés par le langage et qui représente un type d'événement précis.

Événement	Expression	Type de B
Appels de constructeurs	:c--> B	CONSTRUCTOR
Appels de méthodes	:m--> B	METHOD
Appels aux blocs d'initialisation	:b~~> B	BLOCK
Les lancements de 'thread'	:t~~> B	METHOD
Lectures de champs	:r>> B	FIELD
Écritures de champs	:w>> B	FIELD
Les lancées d'exceptions	:x>> B	CLASS
Les exceptions attrapées	:/x\>> B	CLASS

Fig. 6.5: Expression par événement

Dans la figure 6.5, nous prenons soin de distinguer visuellement les événements évoquant des concepts similaires en créant 3 types d'expressions :

1) Ceux composés de '–>' qui représentent les événements qui déplacent (de manière volontaire) le flux d'exécution vers l'entité de programme représentée par la variable à leur droite (ici c'est 'B') tels que les appels de méthodes('m') et les appels de constructeurs('c').

2) Ceux composés de '~~>' faisant référence aux événements non appelés directement dans le code, mais qui se déclenchent suite à un autre événement et qui déplacent également le flux d'exécution vers la variable à leur droite. Nous faisons référence aux appels de bloc d'initialisation ('b') qui sont appelés aux chargement et aux instanciations des classes Jave ainsi qu'aux événements qui démarrent un nouveaux processus d'exécution 'thread'('t').

3) Et ceux composé de '»' qui ne déplacent pas le flux d'exécution vers l'entité de programmation qu'elles référencent tels que la lecture ('r') ou l'écriture('w') d'un champ ou le lancé('x') et la réception d'une exception('/x\').

Sur cette base visuelle, la figure 6.6 nous montre, les événements qui sont considérés par une expression lorsque nous ne mettons pas de symboles ('c','m','b',...) sur la flèche. Nous permettons également l'utilisation d'expressions hybrides en utilisant les caractères '-', '~','»','>'. De la sorte, nous pouvons créer de nouvelles flèches qui regroupent, à la carte, les 3 catégories précédentes. La grammaire attribuée (Annexe A) présente en détail les différentes combinaisons qui sont correcte, mais nous comprenons que lorsqu'une flèche hybride présente la caractéristique visuel d'une des 3 catégories, elle représente cette catégorie. Par exemple, les exemples d'expressions hybrides présentés en figure 6.7 sont corrects :

-Nous constatons en figure 6.7 que nous pouvons également associer les symboles d'événement (Eg. 'm','c',...) avec les expressions hybrides afin de préciser les événements à considérer.
-Afin de bien distinguer le début d'un événement, nous utilisons le double point vertical ' :'.

Événements	Expression	Type de B
Appels de constructeurs Appels de méthodes	:--> B	CONSTRUCTOR METHOD
Appels aux blocs d'initialisation Les lancements de 'thread'	:~~> B	BLOCK METHOD
Lectures de champs Écriture de champs Les lancées d'exceptions Les exceptions attrapées	:>> B	FIELD CLASS

Fig. 6.6: Expressions représentant plusieurs événements de même catégorie

La notion d'appel de bloc est un peu perturbante, mais il est difficile d'en faire abstraction dans une représentation de flux d'exécution. Pour rappel, un bloc d'initialisation peut être déclenché suite à l'appel d'une méthode statique, suite à l'accès à un champ statique ou encore suite à un appel de constructeur. Il est dès lors intéressant de pouvoir l'intégrer dans nos modèles de flux d'exécution afin d'ouvrir la porte aux types de questions suivantes :

1) "Quels sont les événements qui ont déclenché le chargement d'une classe ? (plus concrètement qui ont enclenché l'appel à un bloc d'initialisation)"

:~-» B :b~~> C

ou encore

2) "Quel sont les événements qui déclenchent un bloc d'initialisation qui accède à un champ X"

Événements	Expression	Type de B
Appels de constructeurs Appels de méthodes Lectures des champs Écritures de champs Les lancées d' exceptions Les exceptions attrapées	:-->> B	CONSTRUCTOR METHOD FIELD CLASS
Appels de constructeurs Appels de méthodes Appels aux blocs d'initialisation Les lancements de 'thread' Lectures de champs Écritures de champs Les lancées d'exceptions Les exceptions attrapées	:~-->> B	CONSTRUCTOR METHOD BLOCK FIELD CLASS
Appels de méthodes Lectures de champs	:m,r-->> B	METHOD FIELD

Fig. 6.7: Expressions regroupant plusieurs type de catégories d'événement

```
:~-» B :b~~> C :» X
```

Pour des raisons cosmétiques, nous permettons de fermer le début d'un modèle en ajoutant simplement une lettre majuscule (variable) : `A :~-» B :b~~> C` . Bien entendu dans ce cas, 'A' représente également une entité de programme sur laquelle a eu lieu un événement.

Nous pouvons a présent imaginer plus facilement le flux d'exécution se déplacer au sein du modèle, de variable(entité de programme) à variable(entité de programme).

Pour rappel, dans Java un flux d'exécution se déplace la où sont définis les instructions, c'est à dire dans les méthodes, les blocs d'initialisation et les constructeurs. Dans le dernier exemple, il faut faire attention que si l'événement entre A et B est de type '–>' ou '~~>', le flux d'exécution se déplace de A vers B puis de B vers C. Par contre, si l'événement entre A et B est de type '»', alors le flux se déplacera de A vers C seulement au moment du second événement.

Pour mieux comprendre les limites de notre modèle, nous pouvons comparer la composition de notre modèle de flux d'exécution à la création d'un modèle d'appel sur lequel nous aurions ces quelques règles à savoir.

1)Il faut considérer les expressions se terminant par –> comme étant les briques de base du

modèle sur lequel viennent se greffer les autres types d'expressions. Bien entendu, en réalité, il reste possible de composer des modèles de flux d'exécution sans ces expressions de base.

2)-Les expressions désignant un appel de bloc d'initialisation (:b~~>) peuvent se mettre après n'importe quelle expression représentant un événement qui peut le déclencher. À savoir, l'accès à un champ, appel de méthodes et appel de constructeur. -Les expressions désignant le déclenchement d'une nouveau processus d'exécution (:t~~>) doivent se mettre après un appel de méthode ' :m–>' (qui représentera la méthode 'start()' de la classe 'thread').

3)Les expressions d'accès et de gestion d'exceptions, se terminant par '»', peuvent être greffées à la suite des expressions d'appel '–>' et '~~>' :

Soit directement, pour terminer un modèle comme ceci :

```
|A| :–> |B| :» |C|
```

Ou soit en les attachant à une variable (déjà référencée par un modèle) dans une nouveau modèle séparé par l'opérateur '&'.

```
|A| :–> |B| :–> |C| & |B| :» |C|
```

Nous profitons de cet exemple pour ajouter que dans notre version finale, nous entourons les variables par des barres verticales comme ceci " |C| " afin de marquer la séparation entre les flèches d'événement et les entités de programmation.

Nous remarquons qu'il est possible de composer des modèles en graphe à plusieurs branches. Dans le but de déterminer quelle branche s'exécute avant une autre, nous fournissons des identifiants d'événements qui sont reconnaissables grâce au caractère '@' suivi d'une majuscule. Ces identifiants devront être classés entre eux par les opérateurs (<,>).

```
|A| :–> |B| :{@T1}–> |C| & |B| :{@T2}» |C|
```

Par défaut, nous avons décidé que c'est toujours le modèle déclaré en premier qui s'exécute en premier (ici @T1 avant @T2). Mais si nous voulons exprimer que @T2 s'exécute avant @T1, nous devrons écrire quelque part dans la requête l'expression suivante `'@T2 < @T1'`. Nous reviendrons sur cet exemple dans la section suivante ("Segmentation de la requête en clause") pour expliquer comment intégrer cette dernière expression de comparaison au sein d'une requête.

Pour nous aider à composer un modèle valide, la figure 6.8 nous fournit un tableau nous montrant les successions d'événements qui sont correctes.

:Event1> |B| :Event2> |C|									
Event1	Event2	:c	:m	:t	:b	:x	:/x\	:r	:w
:c		OK	OK	X	OK	OK	OK	OK	OK
:m		OK	OK	OK	OK	OK	OK	OK	OK
:t		OK	OK	X	OK	OK	OK	OK	OK
:b		OK	OK	X	OK	OK	OK	OK	OK
:x		X	X	X	X	X	X	X	X
:/x\		X	X	X	X	X	X	X	X
:r		X	X	X	OK	X	X	X	X
:w		X	X	X	OK	X	X	X	X

Fig. 6.8: Successions possibles d'événements

A ce stade du prototypage, le tableau 6.8 représente principalement une sorte de 'guidelines' simplifiée pour nous aider à comprendre la composition d'un modèle de flux d'exécution. En réalité, si nous devions faire un contrôle rigoureux de la syntaxe pour une version de production, nous devrions prendre en compte les différents sous-types d'éléments de programme définis notamment par les critères d'accessibilité de Java. A défaut de faire ce contrôle, l'utilisateur pourrait avoir des erreurs d'interprétation lorsqu'il emploie le système. La figure 6.9 illustre bien ce problème. Nous y cernons bien la problématique avec la présence d'un domaine de définition de requête plus large de celui réellement interrogé. Dans ces conditions, il est en effet difficile de discerner la raison pour laquelle une requête ne nous renvoie pas de réponse. Est-ce parce que le modèle de requête envoyé au système est mauvais ou est-ce parce qu'il n'y a pas eu d'enregistrement de traces d'exécution par ce système ? L'annexe B définit plus en détail les sous-ensembles d'événements et leurs possibles combinaisons. Dans la suite de ce mémoire, nous ferons l'hypothèse que les mauvaises compositions de requêtes seront contrôlées pour correspondre à ce domaine décrit (Annexe B). Sur base du tableau simplifié (fig.6.8), un test de faisabilité a été mis en place à l'aide d'une grammaire attribuée (Annexe A). A ce niveau de complexité simplifiée, ce test a été réalisé avec succès et nous n'avons pas relevé de remarque particulière.

D'un point de vue sémantique :

La figure 6.10 lie à chaque expression d'événement l'ensemble de valeurs qu'il décrit. L'annexe B construit de manière très détaillée ces ensembles de valeurs.

Étant donné que chacune de nos expressions rentre dans le cadre d'un modèle où chaque événement est précédé d'un autre événement, nous avons créé l'ensemble 'ProgramEntryCallSet' qui désigne un domaine récursif dont le point de départ est la méthode d'entrée du programme. A chaque fois qu'un modèle de flux d'exécution est créé avec une requête, cela représente une séquence qui est un sous-ensemble de cet ensemble (Fig. 6.12).

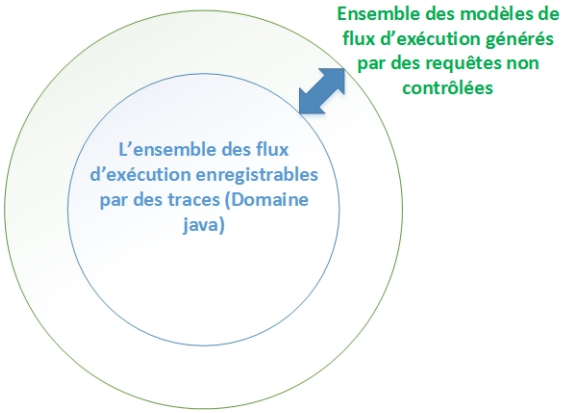


Fig. 6.9: Ensemble de modèles de flux d'exécution non contrôlés

Expression		Domaine sémantique
:c--> |B|	→	BaseConstructorCallSet
:m--> |B|	→	BaseMethodCallSet
:b~~> |B|	→	BaseBlockCallSet
:t~~> |B|	→	BaseThreadCallSet
:r≫ |B|	→	BaseReadFieldSet
:w≫ |B|	→	BaseWriteFieldSet
:x≫ |B|	→	BaseThrowExceptionSet
:/x\≫ |B|	→	BaseCatchExceptionSet
|.| :--> |B|	→	BaseProgramEntryCallSet

Fig. 6.10: Domaine sémantique relié à chaque expression d'événement

Afin de désigner l'appel à la méthode qui constitue le point d'entrée du programme, nous avons créé une nouvelle notation qui consiste a mettre un simple point avant une expression d'appel '\|.\| :–> \|B\|'

Expression		Domaine sémantique
|.| :--> |B| ...	→	ProgramEntryCallSet

Fig. 6.11: Domaine sémantique relié à l'expression désignant la méthode d'entrée d'un programme

Segmentation de la requête en clause

Durant la réalisation de notre langage, nous nous sommes très vite rendu compte qu'il serait de moins en moins facile d'exprimer clairement un modèle de flux d'exécution. En effet, à ce stade, nous n'avons encore que très peu d'information incluse dans le modèle et par conséquent aucun

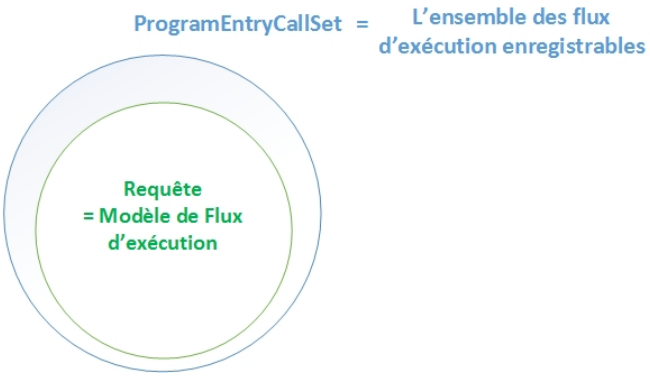


Fig. 6.12: Domaine des flux d'exécution

moyen de filtrer davantage nos événements. Il était devenu évident qui si nous voulions augmenter cette capacité, il nous faudrait indéniablement rajouter beaucoup d'informations telles que des signatures de méthodes, des notions d'héritage et des informations liées au concept dynamique. Pour éviter de surcharger visuellement notre langage, il nous fallait segmenter et répartir nos éléments de requête dans différentes clauses.

Le SQL est probablement le premier langage auquel nous avons pensé. Ne pas s'en inspirer pour structurer notre langage aurait été une erreur. C'est un langage mature, représentant plusieurs années de travail dont la syntaxe est déjà bien connue des développeurs et qui intègre un univers répondant déjà à beaucoup de préoccupations auxiliaires différentes (afficher, agréger, grouper, filtrer). Il possède, en effet, des opérateurs et clauses intéressants que nous pourrions intégrer sans trop de formalisme dans une version de 'production' de notre langage. Nous pensons notamment à "ORDER BY" pour trier les données et "HAVING" pour comparer des résultats de fonction (e.g SUM,AVG,).

Pour notre prototype, nous avons retenu les clauses suivantes :

MATCH : qui permettra de définir des modèles de requête simple comme présenté précédemment. (Inspiration SQL-like : 'Cypher' de Neo4J)

FROM : l'enregistrement des traces d'exécution pouvant regrouper par scénarios d'exécution, la clause 'FROM' permettra de sélectionner les scénarios à prendre en compte. (Inspiration SQL)

WHERE : qui permettra de raffiner le modèle présent dans la clause 'MATCH' en associant des signatures aux différentes variables (lettres majuscules), définir l'ordre d'exécution des événements, définir les relations d'héritage, préciser quelles variables représentent une instance, préciser à quel thread un événement appartient. (Inspiration SQL)

PRINT : 1) permettra de préciser l'information à afficher en listant les variables définies dans les autres clauses. 2) permettra d'appliquer des fonctions d'agrégation sur des variables et d'afficher le résultat. (Inspiration SQL-like : 'QUEL' de OMEGA)

GROUP BY : Permettra de regrouper l'information par type, s'utilise souvent avec des fonctions d'agrégation. (Inspiration SQL)

Sur base de notre dernier exemple, la figure 6.13 nous montre à quoi ressemblerait la requête si nous devions y mettre toutes les clauses.

```
MATCH |A| :--> |B| :{@T1}--> |C| & |B| :{@T2}>> |C|
FROM ...
WHERE @T2 < @T1
GROUP BY ...
PRINT ...
```

Fig. 6.13: Exemple d'une requête composée de toutes les clauses

Pour qu'une requête soit valide, il faut au minimum la présence d'une clause 'MATCH' ou 'PRINT' non-vide. A défaut de clause 'PRINT' ce sera le contenu de la clause 'MATCH' qui sera évalué pour l'affichage (Fig. 6.14).

```
MATCH |A| :--> |B| :{@T1}--> |C| & |B| :{@T2}>> |C|
WHERE @T2 < @T1
```

Fig. 6.14: Exemple d'une requête composée juste des clauses nécessaires

La clause WHERE vient préciser et raffiner le modèle de flux d'exécution construit dans la clause 'MATCH' en rajoutant d'autres informations supplémentaires. La clause 'GROUP BY',quant à elle, ne s'utilise qu'avec la clause 'PRINT'.

A noter que sans les clauses PRINT, GROUP BY et les fonctions d'agrégation, il est difficile d'obtenir un caractère compositionnel élevé à notre langage. En effet, le fait d'obtenir une valeur grâce à un regroupement ou par une agrégation nous offre l'opportunité d'intégrer ce résultat dans une autre requête.

L'information statistique

Seront introduits, dans cette section, 3 types d'informations statistiques capitales pour permettre une formulation plus complète de nos requêtes : nous verrons d'abord les liens d'appartenances entre éléments de programme, puis l'intégration des informations liées aux signatures des éléments de programme, et enfin, les relations d'héritage entre classes/interface.

A) Pour exprimer les liens d'appartenance (Eg CLASS.METHOD.FIELD) entre les différents éléments d'un programme, nous avons choisi d'utiliser le style "Path expression" en proposant l'emploi du point comme opérateur de liaison. L'utilisation du point "." est très intuitive et visuellement adaptée à nos requêtes. De base, une 'expression Path' prend la forme d'une succession de

variables séparées par un point "A.B.C" mais nous verrons plus loin qu'il est possible de remplacer une variable directement par une chaîne de caractères entre guillemet (Eg A.'elemName'.C) ou par un astérisque '*'(quand nous n'avons pas besoin d'une variable à référencer "A.*.C"). La figure 6.15 montre les 3 manières intéressantes d'intégrer ces expressions dans une requête.

```
MATCH |P1.C1.A| :m--> |P2.C2.B|
MATCH |A| :m--> |B|
WHERE P1.C1.A
      & P2.C2.B
MATCH |A| :m--> |B|
WHERE A = P1.C1.X
      & B = P2.C2.Y
```

Fig. 6.15: Intégration d'une expression 'Path' au sein d'une requête

Au sein d'une expression path, chaque variable représente un élément de programmation qui est le parent de l'élément à sa droite (s'il y en a un). Dans le tableau 6.5 nous avions associé un type d'élément de programme par événement (Eg :m–> |B|). Dans une 'path expression', c'est l'élément le plus à droite qui hérite des types implicites que lui impose sa flèche (du moins par défaut). Par exemple dans "|A| :–> |D.C.B|", si B n'a pas de type assigné dans la clause "WHERE" alors il sera, par défaut, de type "METHOD" ou "CONSTRUCTOR". Pour ce qui concerne les types potentiels du parent, ils ont été décrits dans la figure 6.16.

Elément de programme	Valeurs possibles de l'élément de programme parent
CLASS	PACKAGE CLASS INTERFACE BLOCK CONSTRUCTOR METHOD
INTERFACE	PACKAGE INTERFACE CLASS
BLOCK	CLASS
CONSTRUCTOR	CLASS
METHOD	CLASS INTERFACE
FIELD	CLASS INTERFACE METHOD BLOCK

Fig. 6.16: Types possibles pour l'élément parent d'un élément de programme

Par exemple, dans les expressions suivantes, nous comprenons que :

1) dans |A| :m–> |B|,
que B est une méthode et que A est soit une méthode, soit un constructeur, soit un bloc. (Pour rappel, en java se sont les seuls éléments d'où peut partir un appel de méthode)

2)dans |C1.A| :m–> |C2.B|,
C1 et C2 sont les parents respectifs de A et B et peuvent être des classes ou interfaces.

3)dans |P1.C1.A| :m–> |P2.C2.B|
C1 et C2 pouvant être des classes/interfaces internes, P1 et P2 peuvent être de différents types (packages, classes, bloc d'initialisation, constructeurs, méthodes).

Afin de pouvoir définir le type d'un élément de programme en particulier dans une expression 'Path', la figure 6.17 nous offre des opérateurs destinés à être utilisés dans la clause 'WHERE' :

Opérateur	Exemple
ISPACKAGE ()	**MATCH** |P1.C1.A| **:m-->** |P2.C2.B| **WHERE ISPACKAGE** (P1)
ISBLOCK ()	**MATCH** |P1.C1.A| **:m-->** |P2.C2.B| **WHERE ISBLOCK**(P1)
ISINTERFACE()	**MATCH** |P1.C1.A| **:m-->** |P2.C2.B| **WHERE ISINTERFACE** (C2)
ISCLASS ()	**MATCH** |P1.C1.A| **:m-->** |P2.C2.B| **WHERE ISCLASS** (P1)
ISCONSTRUCTOR ()	**MATCH** |P1.C1.A| **:m-->** |P2.C2.B| **WHERE ISCONSTRUCTOR** (A)
ISMETHOD ()	**MATCH** |P1.C1.A| **:m-->** |P2.C2.B| **WHERE ISMETHOD** (A)
ISFIELD ()	**MATCH** |A| **:b~~>** |P2.C2.B| **WHERE ISFIELD** (A)

Fig. 6.17: Opérateurs représentant les éléments de programme

D'un point de vue sémantique :

Dans notre langage, une variable associée à un élément de programme (Eg. B dans ' :–> |B|') est liée à un ensemble de valeurs purement statiques comprenant notamment les informations liées à la signature de cet élément (Annexe B). La figure 6.18 nous liste ces ensembles.

Elément de programme		Domaine de valeurs
PACKAGE	→	PackagePathSet
INTERFACE	→	IntefacePathSet ∪ InnerIntefacePathSet
CLASS	→	ClassPathSet ∪ InnerClassPathSet
CONSTRUCTOR	→	ConstructorPathSet
METHOD	→	MethodPathSet
BLOCK	→	BlockPathSet
FIELD	→	FieldPathSet

Fig. 6.18: Ensemble de valeurs pour chaque élément

Sans vouloir surcharger notre présentation avec les détails de l'annexe, la figure 6.19 nous montre graphiquement la structure 'type' de ces ensembles en prenant comme exemple l'ensemble 'ClassPathSet'. Les autres ensembles de notre figure 6.18 étant composés de la même manière, nous ne les présenterons pas.

Dans notre graphique, nous remarquons que le type 'ClassPathSet' est en réalité composé de beaucoup de sous-ensembles qui sont les résultats de produits cartésiens entre des ensembles 'Custom Path' et des sous-ensembles de l'ensemble 'ClassSet'. Les sous-ensembles de 'ClassSet' représentent les valeurs statiques relatives aux signatures de chaque type de classe Java [1] (Eg 'nameClass','public','final','strictfp',...). Chaque ensemble de 'Custom Path' est le résultat d'une union entre différents sous-ensembles de l'ensemble de la forme 'xxxPathSet' du **parent** direct. Cette forme de récursion crée un domaine qui nous fait remonter jusqu'au premier ensemble de type 'PackagePathSet'(Eg.PACKAGE.PACKAGE.CLASS.METHODE.FIELD).

Dans notre graphique, l'ensemble **"Custom Path"** représente une abstraction qui a été créée pour le besoin de la présentation. Dans notre annexe B, chaque ensemble "Custom Path" qui comporte **plusieurs** "Parent's xxxPathSet" portera un nom qui suit la convention de nommage suivante :**"xxxxxxPath"** (ou une abréviation, Eg 'NASC**P**') et tous les 'Custom path' ne comportant qu'**un seul** ensemble "Parent's xxxPathSet" seront directement représentés par ce dernier ensemble.

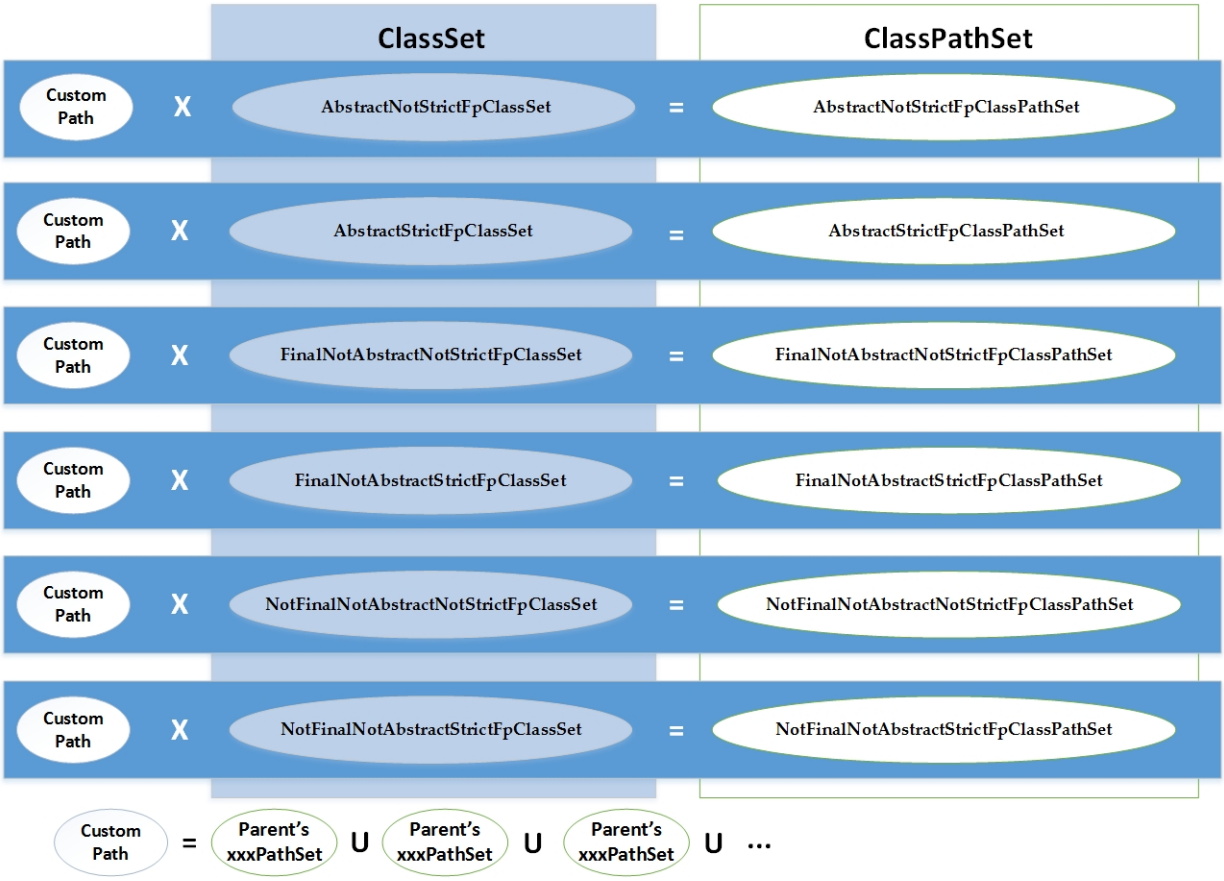


Fig. 6.19: Structure de l'ensemble de valeurs 'ClassPathSet'

Dans notre langage, les variables de nos expressions 'path' peuvent être vues comme une série de points d'ancrages pour définir un sous-ensemble de valeurs de ce domaine récursif. Chaque point d'ancrage venant préciser un ensemble 'xxxSet' (Eg. ClassSet dans le graphique), notamment grâce au référencement de variable dans la clause 'WHERE' de nos requêtes. Ainsi les expressions suivantes sont équivalentes :

- ' :m–> | P2.C2.B | '
- ' :m–> | B | '

A la différence que dans ' | P2.C2.B | ', il est possible de préciser plus en détail le domaine de valeurs de l'expression 'Path' (dans cette exemple c'est l'ensemble MethodPathSet car la flèche impose le type).

B) Sortons de cette vision du domaine pour en revenir à une explication de plus haut niveau. Il est possible de préciser sa recherche en intégrant des informations liées aux différents éléments :

- Pour faire une recherche simplement à partir du nom d'un élément, nous fournissons une technique simple qui consiste à remplacer la variable de l'élément par le nom. Par exemple, pour rechercher tous les appels aux méthodes portant comme nom 'ajouterImage', il suffit d'écrire :

```
|A| :m–> |P.C.'ajouterImage'|
```

Il faut garder à l'esprit qu'avec cette approche, il n'y a plus de variable pour référencer l'élément dans une autre clause. Dans le dernier exemple, cela ne nous pose pas de problème, car le type de l'élément est défini précisément par la flèche qui le précède. Mais, dans le cas contraire, ça peut devenir plus contraignant.

En ce qui concerne le nom des éléments, une attention particulière a été apportée aux blocs d'initialisation, car une classe peut en contenir plusieurs et aucun d'entre eux ne peut être identifié par un nom. Nous avons choisi de les différencier par un numéro représentant leur ordre de déclaration, 1 identifiant celui déclaré en premier, 2 le second, et ainsi de suite...

```
|A| :b~~> |'1'| :b~~> |'2'| :b~~> |'3'|
```

Ce dernier modèle décrit un flux d'exécution se déplacent de bloc d'initialisation à bloc d'initialisation.

- Pour faire des recherches sur plus de critères, nous fournissons (Fig. 6.20) des modèles de signatures statiques.

Type	Signature statique	Exemple
PACKAGE	'*'	**MATCH** |P1.C1.A| **:m-->** |P2.C2.B| **WHERE ISPACKAGE** (P1) **&** P1 **=** 'packageName'
BLOCK	'*'{*\}	**MATCH** |P1.C1.A| **:m-->** |P2.C2.B| **WHERE** P1 = '1'{*\}
INTERFACE	'*'{*}	**MATCH** |P1.C1.A| **:m-->** |P2.C2.B| **WHERE ISINTERFACE** (C2) **&** C2 **=** 'interfaceName'{*}
CLASS	'*'{*}	**MATCH** |P1.C1.A| **:m-->** |P2.C2.B| **WHERE ISCLASS** (P1) **&** P1 **=** 'className'{*}
CONSTRUCTOR	'*'{*(..)/}	**MATCH** |P1.C1.A| **:m-->** |P2.C2.B| **WHERE** A = 'constructorName'{*(..)/}
METHOD	'*'{*(..)»*}	**MATCH** |P1.C1.A| **:m-->** |P2.C2.B| **WHERE** A **=** 'methodName'{*(..)»*}
FIELD	'*'{* *}	**MATCH** |A| **:b~~>** |P2.C2.B| **WHERE ISFIELD** (A) **&** A **=** 'fieldName'{* *}

Fig. 6.20: Modèles de signatures statiques par élément de programme

A l'origine, nous imaginions intégrer nos signatures directement au sein des expressions 'Path' mais comme le démontre l'exemple suivant, cette approche a dû être abandonnée, car cela rendait la lecture de notre langage trop compliquée. C'est notamment cette première contrainte qui nous a poussés à diviser nos requêtes en clauses.

```
|A{* (..)\}| :m–> |P.C{*}.B{* (..)»*}|
```

Il est intéressant de constater (fig. 6.20) que les signatures des blocs d'initialisation, des méthodes et des constructeurs permettent, à eux seuls, de définir le type de ces éléments. Cela est dû au fait que ces signatures possèdent des caractères les distinguant des autres signatures. Pour les méthodes et les constructeurs, il est possible de retirer ce caractère pour obtenir une signature commune '*'{*(..)}. De la sorte, il est possible de l'appliquer sur ' :–>|B|' sans pour autant imposer le type de B.

Au sein de nos signatures, nous remarquons la présence des caractères "*" et "..". L'astérisque signifie que nous voulons toutes les valeurs possibles et le double point est utilisé dans la partie des paramètres pour dire que nous recherchons toutes les entités programme (méthode, constructeur) indépendamment du nombre de paramètres qu'il possède. Par exemple, le système recherchera aussi bien les méthodes sans paramètres que les méthodes avec n paramètres (n>0). Ce double point peut également s'utiliser avec de réelles valeurs de paramètre que nous recherchons.

Par exemple, dans les signatures des méthodes suivantes, nous comprenons que :
1) Dans '*' {* (:int, :bool) »*}, nous recherchons une méthode à 2 paramètres ;

2)Dans '*' {* (.., :int, :bool, ..) »*}, nous recherchons une méthode à deux paramètres qui se suivent avec éventuellement d'autres paramètres avant et après eux ;

3)Dans '*' {* (.., :int, .., :bool) »*}, nous recherchons une méthode qui a deux paramètres pouvant se suivre. Il peut exister, éventuellement d'autres paramètres inconnus avant et après le premier paramètre. Le paramètre ' :bool' doit être le dernier.

Il faut savoir que dans nos signatures chaque type commence par un double point " :" et que le nom complet des types non-primifs doivent être entre guillemets (Eg :'java.lang.object'). Nous verrons plus loin que, dans les signatures dynamiques, le double point nous permettra de distinguer plus facilement les types et les valeurs (qui n'auront pas ce double point).

Les valeurs comprises pour chaque signature sont décrites de manière naïve en figure 6.21. Il est important de savoir que cette description ne précise pas les différentes séquences possibles de 'modifier' pour les types de déclaration et pour les paramètres. Par exemple, Java ne permet pas qu'une même méthode soit 'abstract' et 'final' en même temps. Le domaine (Annexe B) détaille correctement toutes ces contraintes.

Elément de programme	ID (nom)	Type de déclaration	Type des paramètres	Type de retour	Type du champ	Modèle de Signature
Package	1					'1'
Class/interface	1	2				'1' {2}
Method	1	2	3	4		'1' {2 (3) >> :4}
Constructor	1	2	3			'1' {2 (3) /}
Init Block	1	2				'1' {/2/}
Field	1	2			5	'1' {2 5}

Valeurs:
1 : String représentant le nom de l'élément ou un numéro (pour les blocs d'initialisation);
2: Liste de 'modifier'(public, protected, 'friendly', private, static, final, abstract, strictfp, volatile, transient, synchronized, native);
3,4,5: (Liste) de types avec leur 'modifier' . (Types primitifs: char, byte, short, int, long, float, double, boolean, String
non primitifs: le nom complet de classe/interface (avec chemin) ;

Fig. 6.21: Informations comprises par signature statique

C) Les informations sur les relations d'héritage ne sont pas intégrées au sein des signatures des classes/interfaces et ne sont pas intégrées non plus au modèle de flux d'exécution de la clause 'MATCH'. La construction de ces modèles d'héritages se fait au sein de la clause 'WHERE' (Fig. 6.22).

```
MATCH |A| :--> |C.B|
WHERE |'java.lang.Object'| <<e> |C|
```

Fig. 6.22: Requête avec modèle d'héritage

La figure 6.23 nous liste les expressions nous permettant de créer nos arborescences d'héritage. Nous y distinguons à chaque expression une variable "A" à gauche et une variable "B" à droite. C'est à chaque fois la variable "A" qui est étendue/implémentée par la variable "B". Il faut bien entendu faire attention de respecter les règles imposées par Java en la matière. Nous ne pouvons avoir une variable "A" qui étend une variable "A".

Types implicites de A	Expression	Types implicites de B
CLASS INTERFACE	Etend |A| **<<e>** |B|	CLASS INTERFACE
INTERFACE	Implémente |A| **<<i>** |B|	
CLASS INTERFACE	Etend ou Implémente |A| **<<i OR e>** |B|	

Fig. 6.23: Expressions représentant les relations d'héritage

Sur cette base, et de la même manière que pour la construction des modèles de flux d'exécution, il est possible de construire des modèles d'héritage plus longs `|A| «e> |B| «i> |C|`.

> Les éléments positionnés à l'extrémité gauche d'une expression 'Path' (Eg. |**A**.B.C|) peuvent contenir leur nom avec le chemin complet. Pour éviter les incohérences, il faut que ces variables soient positionnées à l'extrémité gauche de toutes les expressions 'Path' dans lesquelles elles sont déclarées (Eg. |A.C| :m–> |A.B|).

L'information dynamique

Seront présentées dans cette section, les signatures dynamiques des événements, le concept d'instance, le concept de 'thread' et enfin le temps d'exécution.

1) Pour exprimer les valeurs des variables, les valeurs des arguments et les valeurs de retour, nous avons créé des signatures dynamiques spécifiques pour chaque type d'événements (Fig. 6.24). Elles sont esthétiquement très similaires aux signatures statiques mais n'acceptent pas le même type de valeurs et ne s'associent pas avec le même type de variables. Ces signatures doivent s'associer avec les variables qui présentent le caractère '@'(Eg ' :@T1–>|B|'). L'exemple en figure 6.25 nous montre cette distinction. Bien entendu, il faut garder une cohérence entre les deux signatures et ne pas se retrouver avec, par exemple, avec une signature statique à 2 paramètres et une

signature dynamique avec 3 valeurs de paramètres

-Dans le suite de cette section, nous verrons que les variables de type '@' (qui sont associées aux flèches) sont à chaque fois utilisées pour référencer l'ensemble des valeurs dynamique associé à un événement. Dans notre annexe B, ces ensembles utilisent la convention de nommage "xxxxDynSet" (Eg 'ThrowExceptionDynSet').
-Il existe une dérogation à cette règle en ce qui concerne le cas des instances, que nous verrons plus loin dans cette section. Dans ce cas, il est plus pratique d'utiliser les variables associés aux éléments de programme pour parler de ce concept(instance)

Evénement	Signature dynamique	
Appel de Méthode	{(..)>>*}	{(..)}
Appel de Constructeur	{(..)\}	
Lecture Champ	{*}	
Ecriture Champ	{OLD.* NEW.*}	
Lancé d'exception	{*}	
Attrape d'exception	{*}	

Fig. 6.24: Signatures dynamiques associées aux événements

```
MATCH  |B| :m{@T1}--> |C|
WHERE  C = 'methodName' {public (:int, :bool) >> :String}
       & @T1 = { ('3', 'true') >> 'helloWorld'}
```

Fig. 6.25: Exemple comportant une signature statique et une signature dynamique

Les valeurs à introduire dans ces signatures sont soit la valeur du paramètre pour tous les types primitifs, soit le type effectif pour les non-primitifs. Toutes les valeurs s'écrivent entre guillemets mais nous distinguons les valeurs pour les types non-primitifs en ajoutant un double point vertical ' :' devant les guillemets (Eg :'java.lang.Object').

2) Notre modèle de requête en flux d'exécution nous a poussés à réfléchir sur le concept d'instance, omniprésent dans ce modèle. Par exemple, dans la requête `|A| :–> |B| :–> |A|`, faut-il admettre de base que les 2 variables 'A' référencent un élément de programme possédant une même instance de classe, ou simplement qu'elles référencent une même signature statique ? Nous avons fini par choisir une composition de requêtes allant du moins restrictif au plus restrictif, de ne pas tenir compte de l'instance de classe liée aux éléments de programmation, sauf si l'utilisateur le précise par un opérateur.

Le cas qui nous a particulièrement poussés dans ce sens a été la détection de "Design pattern". Par exemple, dans la requête simplifiée suivante, cherchant à détecter un modèle M-V-C(modèle-vue-contrôleur), nous avons compris que nous ne voulions pas que cette requête exprime un flux d'exécution passant 2 fois par une même instance de V. A tort, cela aurait réduit considérablement le résultat de recherche.

|V| <–: :–> |C| <–: :–> |M|

-Cet exemple est un modèle de flux d'exécution simplifié pour le besoin de la présentation, notre scénario de test présenté plus loin reprend cette exemple plus en détail.
-Afin de permettre la construction ce type de requêtes, nous avons introduit un nouveau type d'expression. Le composition de ce nouveau type d'expression est beaucoup plus limité comme le montre la figure 6.26. A ce stade-ci, seul une flèche '<–' peut nous permettre d'aller vers la gauche sauf dans le cas d'une expression de fermeture transitive que nous verrons dans la section suivante.
-Par défaut, dans " \|A\| <–{@T1} : :{@T2}–> \|B\| <–{@T3} : :{@T4}–> \|C\| " le flux d'exécution se déplace toujours le plus a droite possible avant de revenir : "@T2 < @T4 < @T3 < @T1", il est toujours possible de redéfinir cette ordre dans la clause 'WHERE'.

|A| <Event2: :Event1>|B|									
Event1	Event2	:c	:m	:t	:b	:x	:/x\	:r	:w
:c		OK*	OK*	X	X	X	X	X	X
:m		OK*	OK*	X	X	X	X	X	X
:t		OK*	OK*	X	X	X	X	X	X
:b		OK*	OK*	X	X	X	X	X	X
:x		X	X	X	X	X	X	X	X
:/x\		X	X	X	X	X	X	X	X
:r		X	X	X	X	X	X	X	X
:w		X	X	X	X	X	X	X	X
*|A|		Constructor	Method	-	-	-	-	-	-

Fig. 6.26: Combinaisons possibles d'événements

Nous fournissons donc les 2 opérateurs (Fig. 6.27) : le premier, pour permettre d'affirmer qu'une variable est liée à une même instance de classe pour tous les événements qui lui sont liés, et le second, pour permettre de définir qu'une instance de classe d'une variable soit équivalente à l'instance de classe d'une autre variable.

Opérateur	exemple
INSTANCEOF()	MATCH |A| :{@T1}--> |B| :{@T2}--> |C| WHERE INSTANCEOF (@T1) = INSTANCEOF (@T2)
ISINSTANCE()	MATCH |V| <--::--> |C| <--::--> |M| WHERE ISINSTANCE(M)

Fig. 6.27: Opérateurs dynamiques liés au concept d'instance de classe

3) Nous l'avons vu, le concept de 'thread' est également présent dans notre modèle d'appel. Grâce à l'opérateur ' :t~~>', il est possible de représenter le déclenchement d'un nouveau processus d'exécution. Plus concrètement, au niveau de notre langage, cela signifie qu'une méthode de type 'start()' à été appelée ce qui engendre un appel à une méthode 'run()'(:m–> \| 'start' \| :t~~> \| 'run' \|).

À la différence que dans ce cas, le flux d'exécution se déplace de ma méthode 'start' vers la méthode 'run' mais le processus d'exécution de la méthode 'start' sera différent du processus d'exécution de la méthode 'run'. L'entité de programme 'run()' hérite d'un nouveau processus

d'exécution.

> Pour rappel, en java, la classe de la méthode 'run' est soit la même que la méthode 'start' soit un objet 'Runnable' différent si la classe de la méthode 'start' a été construite avec un objet de type 'Runnable' en argument [1].

Opérateur	exemple
THREADOF()	MATCH |A| :{@T1}--> |B| :--> |C| :t{@T2}~~> |D| PRINT THREADOF(@T1), THREADOF(@T2)

Fig. 6.28: Opérateurs dynamiques liés au concept de processus d'exécution

4) Dans la partie présentant l'exploitation de traces d'exécution, nous avions fait référence au temps d'exécution. Ce calcul peut se faire à différents niveaux : un scénario, une partie d'un modèle d'exécution ou sur l'exécution d'un bloc d'exécution précis. Les opérateurs suivants permettent de réaliser cette tâche :

Opérateur	exemple
DURATIONOF()	MATCH |A| :{@T1}--> |B| :--> |C| WHERE DURATIONOF(@T1) > 9000 PRINT DURATIONOF(@T1)
STARTTIMEOF()	MATCH |A| :{@T1}--> |B| :--> |C| PRINT STARTIMEOF(@T1
ENDTIMEOF()	MATCH |A| :{@T1}--> |B| :--> |C| PRINT STARTIMEOF(@T1)

Fig. 6.29: Opérateurs dynamiques liés au concept de temps d'exécution

La fermeture transitive

L'expression d'une fermeture transitive est particulièrement indiquée pour factoriser les expressions en arbre d'appel et les relations d'héritage : ce sont des raccourcis d'écriture très intéressants.

En ce qui concerne les expressions d'appel, nous avons nourri le concept en permettant l'introduction d'une limite minimum et/ou maximum de transition. Par exemple, pour afficher les méthodes qui appellent directement ou indirectement une méthode 'x', nous avons la possibilité d'avoir plusieurs niveaux de précision :

1) `|A| :{*}–> |B|` de 0 à l'infini théorique ;
2) `|A| :{2}–> |B|` exactement 2 ;
3) `|A| :{>2,=<4}–> |B|` supérieur à 2 et inférieur ou égale à 4.

- Si nous désirons ajouter également une variable sur la flèche, cela se présente de cette manière ' :{@T1,(*)}–> |B|'. Actuellement il n'y pas d'autres symboles que notre langage permet d'ajouter.
- Dans notre exemple, nous remarquons que l'information sur la relation transitive est définie sur la flèche et non sur la variable 'B', ce qui veut dire que dans le cas du deuxième exemple, cela équivaut à écrire |A| :–> |X| :–> |B| où la variable X n'est pas spécialement équivalente à 'B' (il nous faut exactement 2 appels de méthode ou constructeurs pour arriver à B).
-C'est pour cette raison que nous permettons d'écrire '<-~{*} : dans les expressions à double sens. Pour résumer, notre prototype ne permet pas d'écrire d'autres types de flèche gauche que '<– :' et '<-~{*} :

Nous utilisons la même technique pour les modèles d'héritage.

```
|A| <<i{*}> |B| <<e{3}> |C|
```

Nous profitons de la présentation du concept de fermeture transitive pour introduire les opérateurs (figure 6.30) qui nous permettent de répondre au cas d'utilisation visant à calculer la distance d'appel (ref. section exploitation des traces). La notion a été élargie dans le sens ou le premier opérateur ne prend en compte que les appels aux méthodes et aux constructeurs qui composent le chemin d'exécution entre les deux entités de programme choisies, tandis que le second prend aussi en compte les autres types d'appels.

Opérateur	exemple
STRICTCALLDISTANCE()	MATCH |A| :{@T1}--> |B| :~~> |C| :{@T2}--> |D| PRINT STRICTCALLDISTANCE (@T1, @T2) "==> 2"
CALLDISTANCE()	MATCH |A| :{@T1}--> |B| :~~> |C| :{@T2}--> |D| PRINT CALLDISTANCE (@T1, @T2) "==> 3"

Fig. 6.30: Opérateurs liés au concept de distance d'appel

6.4 Grammaire BNF

Dans l'annexe A, nous présentons une syntaxe concrète (grammaire attribuée) et abstraite de notre langage à l'aide d'une grammaire BNF traditionnelle. Nos syntaxes sont décrites à l'aide de règles qui agrègent ensemble les "token" de notre langage textuel en phrases plus complexes. Une règle est composée d'une "tête" et d'un "corps" qui fourni la définition de la tête : <tête> := corps

Les principales différences entre nos deux grammaires sont que :
-Dans notre grammaire abstraite, il n'y à pas d'attributs et pas de mécanismes de contrôle liés à ces attributs (issues d'un test de faisabilité que nous avons mené à la suite d'une réflexion menée dans le chapitre 6) ;
-Nous n'avons également pas certains caractères encombrants à la lecture telles que certaines parenthèses. Dans la grammaire concrète, nous avons utilisé le simple guillemet pour décrire la barre verticale '|' sinon elle rentrait en conflit avec la barre verticale de la grammaire qui sert à créer des alternatives dans une règle ;

-Les formes de récursion traditionnelles d'une grammaire BNF stricte ont été remplacées par des opérateurs représentant des répétitions "*"(>=0) et "+"(>0) ;

6.5 Scénario de validation

Pour notre test, nous avons choisi de définir un scénario impliquant, tout d'abord une interaction avec un programme représentant la phase d'acquisition de traces, suivit d'une phase de questionnement impliquant une trentaine de questions à exprimer dans notre langage. Tout au long du scénario impliquant nos questions, nous continuerons à introduire de nouveaux opérateurs lorsque cela est nécessaire. Ils seront relevés dans la colonne des notes.

Pour motiver notre exemple, nous imaginons vouloir explorer, modifier et corriger un programme de géométrie du style Géogebra [5]. C'est un programme Open Source téléchargeable gratuitement qui nous sert principalement de référence de style. Bien entendu, aucun des problèmes énumérés par la suite n'est présent dans ce programme. Ce scénario est une fiction dans laquelle aucune réelle interaction avec un système est réalisée.

Comme objectifs, nous imaginons vouloir changer la couleur des points bleus affichés dans le graphique en rouge, rechercher la cause d'un ralentissement lorsque nous créons une droite, et découvrir pourquoi l'utilisation de la mémoire du programme augmente fortement quand nous déplaçons un point vers une autre coordonnée.

En figure 6.31 est présenté le scénario d'utilisation qui permet d'enregistrer les traces d'exécutions.

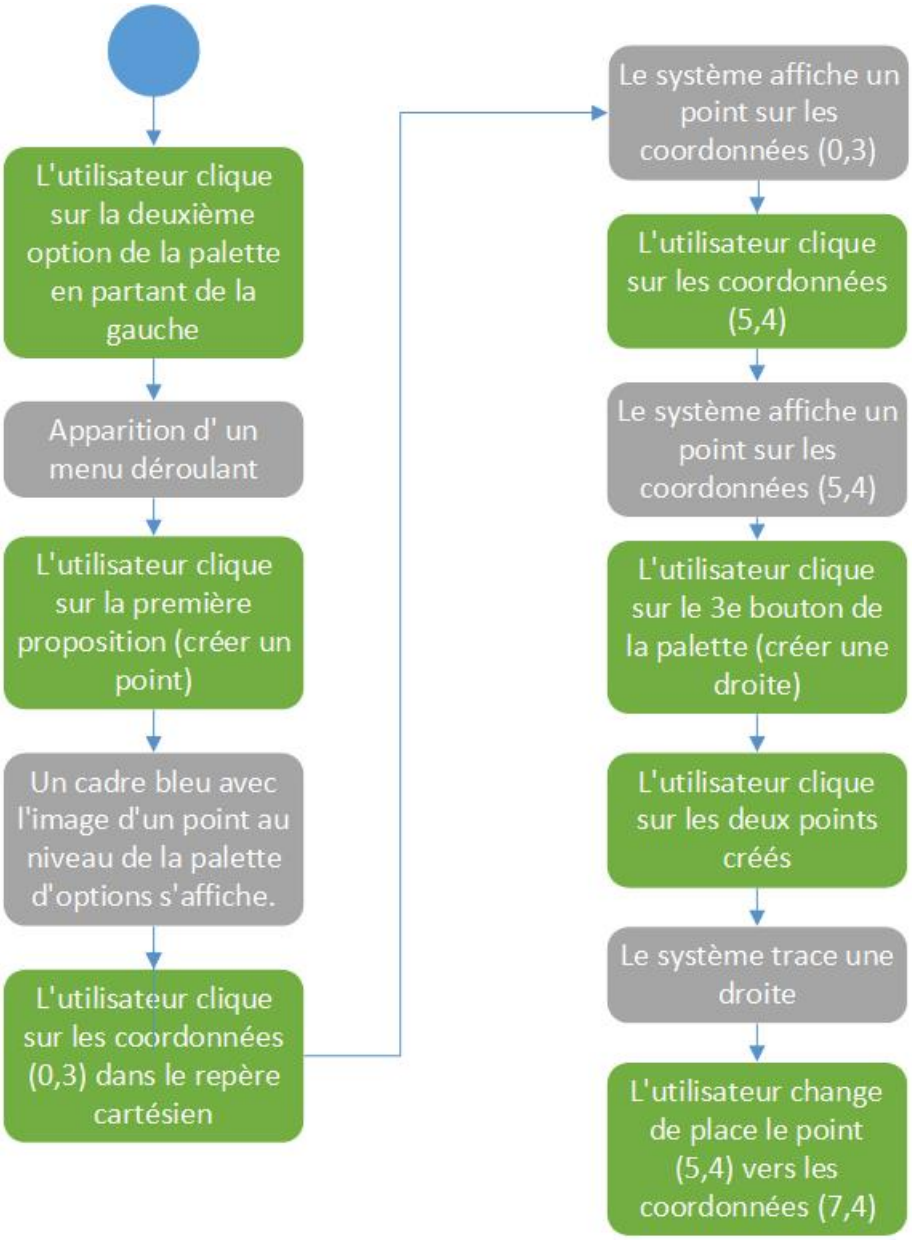


Fig. 6.31: Scénario d'utilisation pour créer une droite

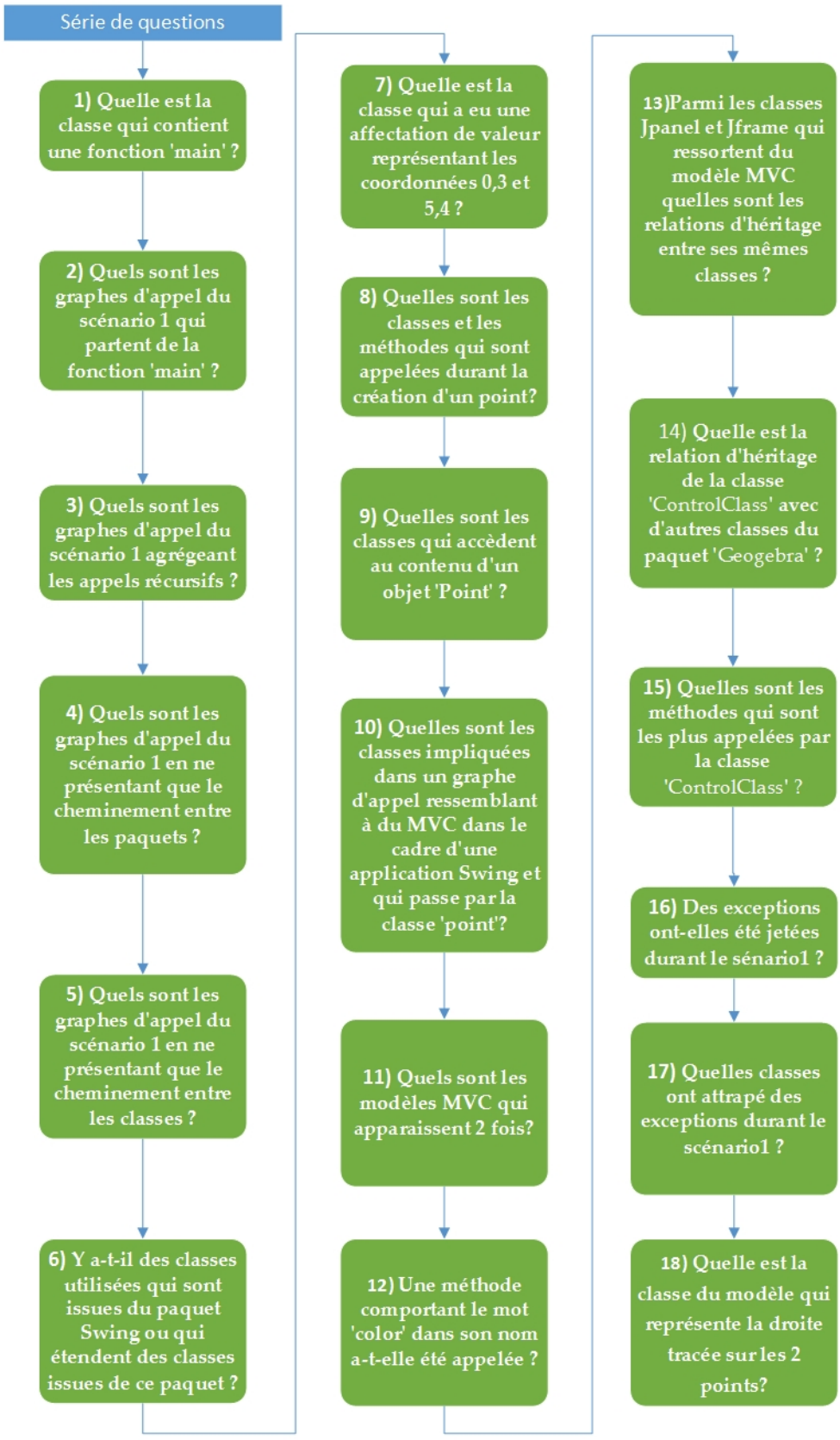
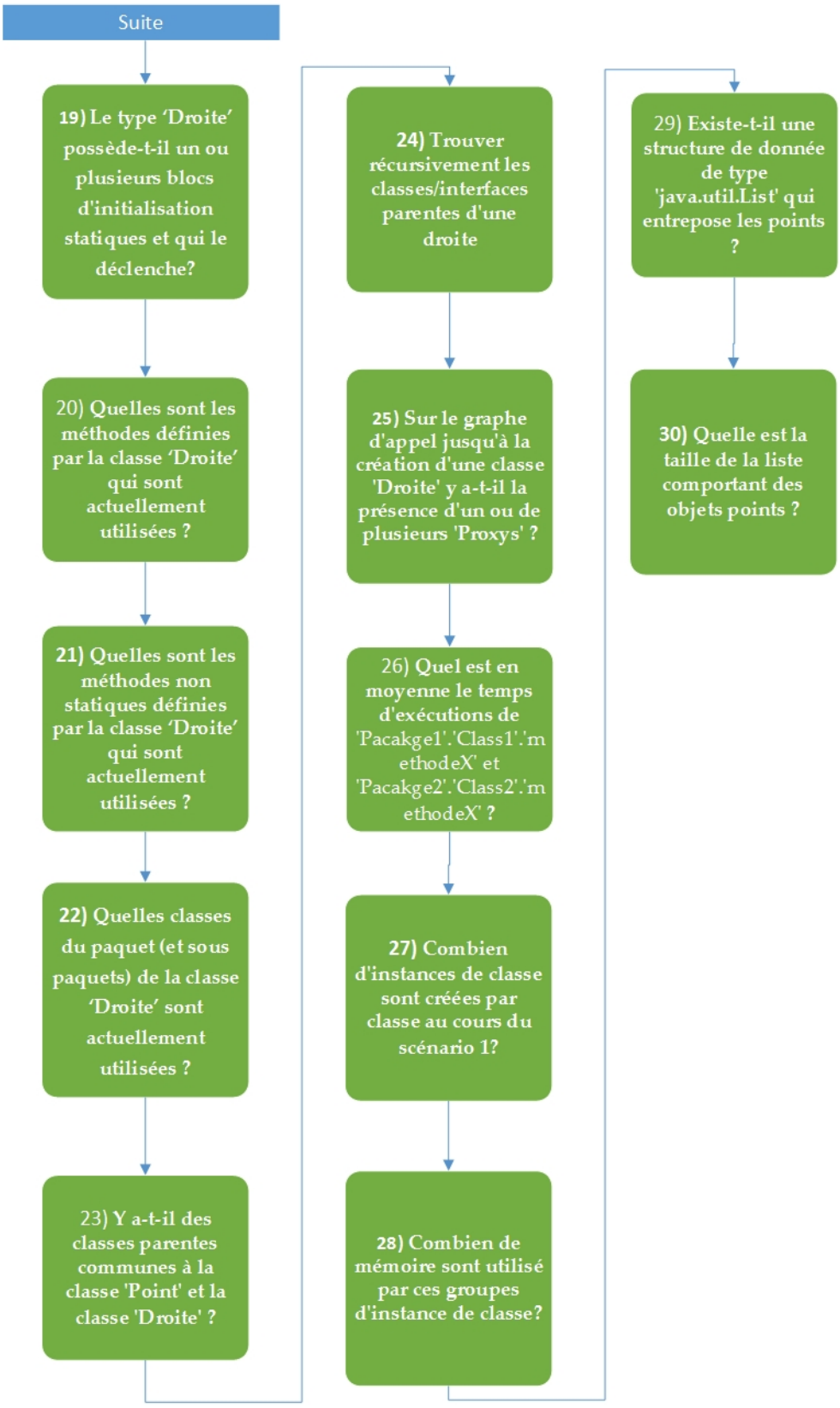


Fig. 6.32: Série de questions


Suite
19) Le type 'Droite' possède-t-il un ou plusieurs blocs d'initialisation statiques et qui le déclenche?
20) Quelles sont les méthodes définies par la classe 'Droite' qui sont actuellement utilisées ?
21) Quelles sont les méthodes non statiques définies par la classe 'Droite' qui sont actuellement utilisées ?
22) Quelles classes du paquet (et sous paquets) de la classe 'Droite' sont actuellement utilisées ?
23) Y a-t-il des classes parentes communes à la classe 'Point' et la classe 'Droite' ?
24) Trouver récursivement les classes/interfaces parentes d'une droite
25) Sur le graphe d'appel jusqu'à la création d'une classe 'Droite' y a-t-il la présence d'un ou de plusieurs 'Proxys' ?
26) Quel est en moyenne le temps d'exécutions de 'Pacakge1'.'Class1'.'methodeX' et 'Pacakge2'.'Class2'.'methodeX' ?
27) Combien d'instances de classe sont créées par classe au cours du scénario 1?
28) Combien de mémoire sont utilisé par ces groupes d'instance de classe?
29) Existe-t-il une structure de donnée de type 'java.util.List' qui entrepose les points ?
30) Quelle est la taille de la liste comportant des objets points ?

	Questions		Notes
- Pour commencer, nous voulons trouver le point d'entrée du programme.			
1	**Quelle est la classe qui contient une fonction 'main' ?**	**MATCH** |.| :m--> |X| **FROM SCENARIO** 'scénario1' **WHERE** X = C.'main' {public static (:String[]) >> :void } **PRINT** C	Nouveauté: |.|:m--> |A| signifie qu'il n'y a pas d'appel avant l'appel de A et que A doit être la première méthode appelée du programme.
- Ensuite, nous voulons découvrir un peu le programme.			
2	**Quels sont les graphes d'appel du scénario 1 qui partent de la fonction 'main' ?**	**MATCH** |.| :m--> |X| :{*}--≫ |X2| **FROM SCENARIO** 'scénario1' **WHERE** X = 'main' {public static (:String[]) >> :void }	
- Nous imaginons recevoir beaucoup de résultats peu lisibles et nous voudrions alléger la présentation en appliquant des fonctions d'agrégation.			
3	**Quels sont les graphes d'appel du scénario 1 agrégeant les appels récursifs ?**	**MATCH** [|.| :m--> | X| :{*}--≫ |X2|] **AS** P **FROM SCENARIO** 'scénario1' **WHERE** X = 'main' {public static (:String[]) >> :void } **PRINT** !REQ(P) -->agrégation post-traitement	Remarque: DISTINCT de SQL agit sur l'ensemble des résultats tandis que !REQ agit sur chaque résultat séparément pour les appels récursifs

Fig. 6.33: Scénario

4	**Quels sont les graphes d'appel du scénario 1 (en ne présentant que le cheminement entre les paquets) ?**	**MATCH** [|.| :m--> |X| **:{*}~->** |X2|] **AS** P **FROM SCENARIO** 'scénario1' **PRINT** PACKAGEVIEW(P)	Nouveauté: - PACKAGEVIEW
5	**Quels sont les graphes d'appel du scénario 1 (en ne présentant que le cheminement entre les classes) ?**	**MATCH** [|.| :m--> |X| **:{*}~->** |X2|] **AS** P **FROM SCENARIO** 'scénario1' **PRINT CLASSVIEW** (P)	Nouveauté: - CLASSVIEW
- Le résultat global reste difficile à comprendre avec beaucoup de classes résultantes. Nous nous tournons vers une analyse plus spécifique à la réalisation d'interfaces. Nous voudrions savoir si le programme utilise des classes issues du paquet Swing de 'javax'.			
6	**Y a-t-il des classes utilisées qui sont issues du paquet Swing ou qui étendent des classes issues de ce paquet ?**	**FROM SCENARIO** 'scénario1' **WHERE** 'javax.swing.*' **<<e or i>** |C| **PRINT** C	
- Nous constatons que certaines classes étendent les classes 'javax.swing.JPanel' et 'javax.swing.JFrame' ce qui nous confirment que nous sommes en présence d'une application de type Swing. - Nous cherchons maintenant à découvrir la classe du modèle qui représente un point et découvrir ce qui se passe autour de cette classe.			
7	**Quelle est la classe qui a eu une affectation de valeur représentant les coordonnées 0,3 et 5,4 ?**	**MATCH** |*| **:w{@T1}>>** | C1.X1| **&** |*| **:w{@T2}>>** | C1.Y1| **FROM SCENARIO** 'scénario1' **WHERE** (@T1{'0'} **&** @T2{'3'}) **OR** (@T1{'5'} **&** @T2{'4'}) **&** ISINSTANCE(C1) **PRINT** C1	Remarque: Ici, l'assignation de 0 peut être de type entier ou String

- Nous découvrons une classe portant le nom 'point'			
8	**Quelles sont les classes et les méthodes qui sont appelées durant la création d'un point?**	**MATCH** |*| :c--> |'Point'.*| :m--> |C.M| **MATCH** |*| :c--> |'Point'.*| :b{*}~~> |*| :m--> |C.M| **FROM SCENARIO** 'scénario1' **PRINT DISTINCT(C.M)**	
9	**Quelles sont les classes qui accèdent au contenu d'un objet 'Point' ?**	**MATCH** |C1| :m,r,w--≫ |C2.X| **FROM SCENARIO** 'scénario1' **WHERE ISINSTANCE** (C2) **& C2** = 'xxx.Point' **& ISCLASS**(C1) **PRINT** C1	
- Suite à notre découverte d'une classe portant le nom 'point', nous cherchons à découvrir une structure MVC			
10	**Quelles sont les classes impliquées dans un graphe d'appel ressemblant à du MVC dans le cadre d'une application Swing et qui passent par la classe 'point'?**	**MATCH [** |V| **<-~{*}::{*}~->** |C| **<-~{*}::{*}~->** |M| **]** as P **FROM SCENARIO** 'scénario1' **WHERE (** |'javax.swing.*'| **<<e>** |V|) **& ISCLASS**(M) **& M** = 'Point' **PRINT CLASSVIEW(DISTINCT** (p))	

- Pour améliorer notre recherche, nous décidons de tenir compte du fait que nous avons créé 2 points exactement et que donc un cheminement lié à la création de ces points devrait être présent 2 fois.			

11	**Quels sont les modèles MVC qui apparaissent 2 fois?**	`MATCH [ |V| <-~{*}::{*}~-> |C| <-~{*}::{*}~-> |M| ] as P`
		`FROM SCENARIO 'scénario1'`
		`WHERE ( | 'javax.swing.*' | <<e> |V| )`
		`& ISCLASS(M)`
		`& M = 'Point'`
		`GROUP BY P`
		`HAVING COUNT(P) = 2`
		`PRINT CLASSVIEW(P),COUNT(P)`
			Nouveauté :
			-GROUP BY
			-HAVING COUNT

- Parmi les classes vues du modèle MVC, nous cherchons à identifier la méthode qui règle la couleur.			

12	**Une méthode comportant le mot 'color' dans son nom a-t-elle été appelée au sein des classes de la vue?**	`MATCH |*|:m-->|M|`
		`WHERE M = '*Color*'`
		`& {M} WITHIN (`
		`MATCH |V| <-~{*}::{*}~-> |C| <-~{*}::{*}~-> |M|`
		`FROM SCENARIO 'scénario1'`
		`WHERE |'javax.swing.*'| <<e> |V|`
		`GROUP BY P`
		`HAVING COUNT(P) = 2`
		`PRINT V )`
			Nouveauté :
			- WITHIN

<table>
<tr><td colspan="4">- Nous trouvons la classe 'VPoint' sur laquelle une méthode 'setColor()' a été appelée. Nous sommes à présent capable de modifier la couleur du point.
- Nous décidons d'en profiter pour découvrir également les relations d'héritage entre les classes de la partie visualisation qui sont ressorties de notre dernière recherche du modèle MVC</td></tr>
<tr><td>13</td><td>Parmi les classes Jpanel et Jframe qui ressortent du modèle MVC, quelles sont les relations d'héritage entre ces mêmes classes ?</td><td>FROM SCENARIO 'scénario1'
Where [|V1| <<i OR e> |V2|] AS S
& {V1, V2} WITHIN (
MATCH DISTINCT(|V| <-~{*}::{*}~-> |C| <-~{*}::{*}~-> |M|)
FROM SCENARIO 'scénario1'
WHERE ('javax.swing.JPanel' <<e> |V|
OR 'javax.swing.JFrame' <<e> |V|)
& ISCLASS(M)
& M = 'Point'
PRINT V)
PRINT DISTINCT(S)</td><td></td></tr>
<tr><td colspan="4">- Nous avons remarqué que les classes du programme se situent principalement dans un paquet nommé 'GeoGebra', nous voudrions également étudier les relations d'héritage qui existent autour d'une classe de contrôle 'ControleClass' que nous avions découvert précédemment au sein de ce paquet.</td></tr>
<tr><td>14</td><td>Quelle est la relation d'héritage de la classe 'ControlClass' avec d'autres classes du paquet 'Geogebra' ?</td><td>FROM SCENARIO 'scénario1'
WHERE |P..C| <<i or e*> |'xx.xx.ControlClass'|
OR |'xx.xx.ControlClass'| <<e*> |P..C|
& ISPACKAGE(P)
& P = 'xx.xx.geoGebra'
PRINT C</td><td>Nouveauté :
- ..
- Quand aucune autre variable n'est déclarée avant, nous devons pouvoir mettre tout le chemin.</td></tr>
</table>

15	**Quelles sont les méthodes qui sont les plus appelées par la classe?**	**MATCH** |C| **:m-->** |M| **FROM SCENARIO** 'scénario1' **WHERE ISCLASS**(C) & C = 'xx.xx. ControlClass' **GROUP BY** M **ORDER BY DESC COUNT**(M) **PRINT** M, **COUNT**(M)	Nouveauté : -ORDER BY -DESC / ASC
- Maintenant que nous avons fini notre première modification, nous allons explorer ce qu'il se passe autour de la création d'une droite et rechercher la cause d'un ralentissement.			
16	**Des exceptions ont-elles été jetées durant le scénario 1 ?**	**MATCH** |*| **:x>>** |E1| **FROM SCENARIO** 'scénario1'	
17	**Quelles classes ont attrapé des exceptions durant le scénario 1 ?**	**MATCH** |*| **:/x\>>** |E1| **FROM SCENARIO** 'scénario1'	
18	**Quelle est la classe du modèle qui représente la droite tracée sur les 2 points?**	**MATCH** |*| **:c,m-->** |X| **FROM SCENARIO** 'scénario1' **WHERE X** = C.'set*' {(..,:'xxx.Point',..) >> *} **OR X** = C. ' *' {(..,:'xxx.Point',:'xxx.Point',..)/} **PRINT** C	

19	**Le type 'Droite' possède-t-il un ou plusieurs bloc(s) d'initialisation statique et qui le déclenche?**	**MATCH** |*| :-->> |*| :b{*}~~> |X.B| **FROM SCENARIO** 'scénario1' **WHERE** X = 'xx.xx.Droite' B = ' * ' {\ static \}	
20	**Quelles sont les méthodes définies par la classe 'Droite' qui sont actuellement utilisées ?**	**MATCH** |*| :m--> |'Droite'.M| **FROM SCENARIO** 'scénario1' **PRINT distinct** (M)	
21	**Quelles sont les méthodes non statiques définies par la classe 'Droite' qui sont actuellement utilisées ?**	**MATCH** |*| :m--> |'Droite'.M| **FROM SCENARIO** 'scénario1' **WHERE** M = '*' {!static (..)>> * } **PRINT distinct** (M)	Nouveauté : -!static
22	**Quelles classes du paquet (et sous paquets) de la classe 'Droite' sont actuellement utilisées ?**	**MATCH** |A|:~->> |*| **FROM SCENARIO** 'scénario1' **WHERE** P.'Droite'.* **& ISPACKAGE**(P) **&** X = P **&** A = X..B **PRINT distinct**(A)	

23	**Y a-t-il des classes parentes communes à la classe 'Point' et la classe 'Droite' ?**	**FROM SCENARIO** 'scénario1' **WHERE** |C1| **<<e OR i>** |'xxx.Point'| **&** |C2| **<<e OR i>** |'xxx.Droite'| **&** C1 = C2 **PRINT** C1,C2	
24	**Trouver récursivement les classes/interfaces parentes d'une droite**	**FROM SCENARIO** 'scénario1' **WHERE** |C| **<<e* OR i>** |'xxx.Droite|	
25	**Sur le graphe d'appel jusqu'à la création d'une classe 'Droite' y a-t-il la présence d'un ou de plusieurs 'Proxys' ?**	**MATCH** |*| :{*}--> |C1.A| :--> |C2.A| :c--> |'Droite'| **FROM SCENARIO** 'scénario1' **WHERE** A = M {*(..,T1,T2,..)>>*} **PRINT** C1,C2	Nouveauté: Rajouter des variables dans les signatures nous permettra d'appliquer des opérateurs de comparaison pour variables. ex : même Type, même Valeur, même nombre de paramètres, même signature.
- Nous constatons qu'il existe effectivement 2 méthodes portant le même nom qui s'appellent successivement, mais qui ne viennent pas de la même classe et du même paquet, mais dont l'une étend la seconde . 'Pacakge1'.'Class1'.'methodeX' et 'Pacakge2'.'Class2'.'methodeX'. Nous pensons alors à comparer les temps d'exécutions des 2 méthodes.			
26	**Quel est en moyenne le temps d'exécutions de** 'Pacakge1'.'Class1'.'methodeX' **et** 'Pacakge2'.'Class2'.'methodeX **?'**	**MATCH** |*| :{@T1,(*)}--> |C1.A| :{@T2,(*)}--> |C2.A| :c--> |'Droite'| **FROM SCENARIO** 'scénario1' **WHERE** A = 'methodeX' {*(..,T1,T2,..)>>*} & C1 = 'Package1.Class1' & C2 = 'Package2.Class2' **GROUP BY** C1.A, C2.A **PRINT** C1.A, AVG(DURATIONOF(@T1)), C2.A ,AVG(DURATIONOF(@T2))	

- Nous constatons que la méthodes 1 met 4 secondes pour s'exécuter tandis que l'autre se termine presque instantanément. En regardant les commentaires du code, nous constatons qu'il s'agit bien d'une classe test proxy servant à enregistrer les informations sur les coordonnées et que cette classe n'a pas été retirée pour la version de production. Nous décidons de la retirer, ce qui règle le problème du ralentissement. - Nous allons à présent rechercher pourquoi la mémoire prend plus de place lorsque nous déplaçons simplement un point sur l'écran.			
27	**Combien d'instances de classes sont créées par classe au cours du scénario 1?**	**MATCH** |*|:c{@T1}-->|A| **FROM SCENARIO** 'scénario1' **WHERE ISCLASS**(A) **GROUP BY A** **PRINT A, COUNT**(@T1)	Remarque : Approximation. Cela ne tient pas compte des instances qui ne sont plus référencées et qui sont détruites par le ramasse-miettes
- Nous observons une grande quantité d'instances de la classe Point son créer.			
28	**Combien d'espace mémoire est utilisé par ces groupes d'instance de classes?**	**MATCH** |*|:c{@T1}-->|A| **FROM SCENARIO** 'scénario1' **WHERE ISCLASS**(A) **GROUP BY A** **PRINT A, COUNT**(@T1) x MEMORYOF(A)	Nouveauté : -MEMORYOF
- Nous nous rendons compte que l'espace mémoire utilisé par les instances de classe Point correspond à la quantité de la fuite de mémoire. Nous nous demandons si les instances de la classe Point sont bien détruites.			
29	**Existe-t-il une structure de donnée de type 'java.util.List' qui entrepose les points ?**	**MATCH** |*| **:m-->** |C.M| **FROM SCENARIO** 'scénario1' **WHERE** |'java.util.List' | **<< i >>** |C| & M = 'get'{* (..)≫:'xx.xx.Point'} **PRINT** C	

- Nous nous posons la question de savoir quelle est la taille à cette liste ?			
30	**Quelle est la taille de la liste comportant des objets points ?**	**MATCH** |*| **:m{@T1}-->** |C.'size'| **FROM SCENARIO** 'scénario1' **WHERE** {C} **WITHIN (** **MATCH** |*| **:m-->** |C.M| **FROM SCENARIO** 'scénario1' **WHERE** |'java.util.List' | **<< i >** |C| & M = 'get'{* (..)≫ :'xx.xx.Point'} **PRINT** C **)** **& @T1 = {(..)≫ S}** **PRINT STARTTIME(@T1) ,S**	
- Nous nous rendons vite compte que la liste atteint des tailles de plus en plus grandes. Ce qui nous laisse à penser qu'à chaque déplacement, le système crée une nouvelle instance de Point qui est stockée dans une liste sans détruire l'ancienne instance de Point. Nous avons trouvé notre fuite de mémoire.			

7. LIMITES ET ÉVOLUTIONS DU LANGAGE

Dans cette partie, nous montrerons que notre prototype de langage a déjà une capacité d'évolution très acceptable. Quand nous parlons de capacité d'évolution, il s'agit surtout d'offrir un langage qui permet la création d'une version suivante sans devoir changer fondamentalement toutes les bases déjà acquises. Cette caractéristique est très importante, car des évolutions se feront tout au long de son cycle de vie. Comme preuve, en validant notre prototype avec un scénario, nous avons déjà repéré quelques limites et évolutions intéressantes :

Évolution des expressions 'Path'

1) Afin de perfectionner la recherche par nom, il est tout a fait possible d'intégrer des concepts d'expressions régulières pour rechercher les noms commençant, comprenant ou se terminant par un mot. Il faut juste faire attention au symbole à utiliser, notamment avec le symbole $ qui peut être utilisé dans le nom d'une Classe. Par exemple AspectJ utilise '*' pour réaliser cette tâche.

Appliqué à notre langage, cela donne :

- `|A| :m–> |'ajouter*'|` pour rechercher tous les appels de méthodes dont le nom commence par le mot 'ajouter'

- `|A| :m–> |'*ajouter*'|` pour rechercher tous les appels de méthodes comprenant le même mot 'ajouter' dans leur nom.

- `|A| :m–> |'*ajouter'|` pour rechercher tous les appels d'une méthode, dont le nom se termine par le mot 'ajouter'

Cette évolution peut être également appliquée à toutes les valeurs de type 'String' au sein des signatures.

2) Nos expressions 'path' ne permettent actuellement pas d'exprimer un nombre d'entités variable entre 2 informations. Il peut, en effet, arriver que l'on connaisse seulement le nom d'un paquet et le nom d'une méthode, mais que nous ne sachions pas très bien à quel niveau de profondeur se situe la méthode par rapport au paquet. Dans quel sous-paquet se trouve la classe contenant la méthode et à quel endroit est déclarée cette classe. S'agit-il d'une classe interne à une classe ou à un bloc ?

-Avec l'expression, `|P1.C1.A| :m–> |'nomDePaquet'.C2.'nomDeMéthode'|`, nous limitons la recherche à 1 niveau de profondeur.

-À même titre que |P1.C1.A| :m–> |'nomDePaquet'.C2.X1.'nomDeMéthode'| limite la recherche à la profondeur de niveau 2, éliminant donc la profondeur 1 et les profondeurs supérieures à 2.

Le caractère '*' étant déjà proposé dans le point précédent, nous en avons recherché un nouveau pour permettre d'exprimer une relation transitive (>=0). À nouveau, AspectJ nous a inspirés. Afin d'élargir l'ensemble d'inclusion de l' expression de ses paquets, il permet le deux points comme séparateur '..' entre le nom d'une classe et le nom d'un paquet. Dans le cadre d'une l'instrumentation AspectJ, cela signifie qu'il faut instrumenter toutes les classes portants un certain nom se trouvant dans un certain paquet et tous ses sous-paquets.

Dans un sens un peu différent, mais dans le même ordre d'idée, nous pourrions utiliser ce double point pour exprimer une relation transitive de profondeur.

```
|P1.C1.A| :m–> |'nomDePaquet'..X1.'nomDeMéthode'|
```

3) Dans certains cas, la forme rapide de recherche de classe par nom que nous proposons peut être fortement contraignante. Le nom remplaçant la variable dans l'expression path, il n'est plus possible de référencer cette entité classe dans la clause 'WHERE' afin d'y apporter des compléments d'information, notamment sur ses relations d'héritage. Le caractère '+' est un caractère interdit dans le nom d'une classe et n'a pas encore été utilisé jusqu'à présent. De ce fait, sans introduire de confusion, il est tout à fait possible de le rajouter en fin de mot pour exprimer une relation transitive d'héritage. Cette évolution ouvre la porte à un nouveau type de formulation de requête :

```
|'service'.'IMailService+'| :m–> |B|
```

Cette requête veut dire, "Donne moi tous les appels de méthode qui partent des classes qui implémentent l' interface 'IMailService' du paquet service".

Dans la clause 'PRINT'

4) Afin de répondre de manière plus minutieuse à des préoccupations d'affichage, nous pourrions imaginer référencer une partie du modèle dans la clause PRINT. Notre modèle d'appel pouvant être décomposé avec l'opérateur '&', il est tout à fait possible d'isoler une partie du modèle et de lui assigner une variable grâce à l'opérateur 'AS'. Cette variable représenterait cette section du modèle dans la clause PRINT.

```
MATCH |D.B| & [|B| :c–>|A|] AS M & |A.C|
PRINT M
```

5) L'intégration d'une clause PRINT est également très pratique pour introduire de nouvelles fonctionnalités dans la requête. C'est un peu comme un terrain de jeu dans lequel il est possible d'appliquer des fonctions et des opérations comme bon nous semble.

Par exemple, dans cet article [10], l'auteur parle des possibilités offertes par une instrumentation dynamique pour calculer le nombre d'instances d'une classe et la taille que peuvent prendre ces instances en terme de mémoire. Pour ajouter l'option d'afficher le nombre d'instances d'une

classe à un moment donné, nous pourrions simplement rajouter un opérateur NBINSTANCEOF() que nous appliquerions à une variable du modèle d'appel.

De manière similaire, pour obtenir combien de place mémoire prend toutes les instances d'une classe ou une instance en particulier, nous pourrions définir la fonction 'MEMORYOF()' qui fournirait la taille que prend une instance ou un groupe d'instances en mémoire.

De même que sur ces résultats, nous pourrions appliquer des opérations arithmétiques classiques pour obtenir des pourcentages.

```
MATCH [|B| :c–>|A|] AS M
PRINT M, MEMORYOF(INSTANCEOF(A)) / MEMORYOF(ALLINSTANCEOF(A))
```

8. CONCLUSION GÉNÉRALE

En conclusion, nous pouvons dire que nous avons réussi à mettre en place un prototype de langage capable d'interroger des traces d'exécution. Le défi étant de fournir un langage utilisable pour tous les développeurs Java, afin qu'ils puissent remplir leurs besoins en terme d'interrogation.

Notre approche fut de commencer par parcourir différents types de langages existant dans le domaine de l'interrogation de code source et de relever dans la littérature scientifique un maximum de remarques pertinentes concernant ces langages et leur conception.

Cette approche nous a permis non seulement d'améliorer notre vue d'ensemble des différentes possibilités de langage, mais également d'acquérir l'expérience nécessaire à une bonne réalisation de notre prototype. Par exemple, nous avons pu, de la sorte, prendre conscience des risques que peut apporter le choix d'un langage trop permissif tels que les langages naturels, tout comme découvrir des limites cachées que peut engendrer le choix de certaines approches de langages structurés, tel que le SQL ou le prolog.

Bien entendu, nous avons pris soin de rester objectifs face aux critiques pour ne pas, par exemple, faire d'amalgame entre une critique sur la faiblesse précise d'un langage et une autre portant sur le langage en général. Pour illustrer cela, la critique concernant la difficulté d'exprimer des requêtes récursives avec les langages de type SQL ne nous a pas empêchés de nous inspirer du SQL pour sa structure intéressante.

D'un point de vue de l'approche utilisée pour l'étude du domaine, bien que le 'code querying' aborde principalement, mais pas exclusivement, l'aspect statique, nous pouvons dire que commencer par son étude nous a été plutôt bénéfique. D'un point de vue des terminologies utilisées, le code querying nous a poussés à créer une ontologie de langage très proche du développeur en utilisant des concepts directement présents dans le code source, alors qu'une étude uniquement basée sur l'aspect dynamique nous aurait probablement poussés à produire un langage plus pauvre et limité en capacité d'expression. C'est important de le souligner, car nous pouvons dire que c'est cette approche qui nous a finalement influencés à mettre en place un langage mixte où les éléments statiques sont considérés comme des entités vivantes laissant des traces de leurs activités. C'est certainement un critère de qualité pour notre prototype qui permet dès lors un plus haut niveau d'expressivité.

Le principe d'itération, proposé par la méthodologie pour construire correctement notre langage, a répondu positivement à nos attentes. Pour résumer, cela nous a permis de partir d'une petite idée de départ et de construire étape par étape notre langage. Le résultat est un langage au

style moins généraliste et imprégné de différentes inspirations qui ont été à chaque fois choisies pour répondre à un type de préoccupation particulière.

Nous avons réussi à atténuer le risque de complexité qui était décrit dès le début par notre méthodologie en appliquant les conseils de B. Tognazzini à chaque itération. À défaut d'avoir pu réaliser une étude quantitative sur un échantillon de personnes pour évaluer notre langage au cours de sa création, nous nous retrouvons tout de même avec un langage dont la mixité de style est restée balisée par des principes et des règles (cohérence et utilisation de métaphores).

Nous avons également montré que notre langage était capable d'intégrer correctement de nouvelles fonctionnalités lors de la réalisation d'un scénario de test. Dans notre cas, "correctement" signifie que nous n'avons pas eu le besoin de modifier les bases de notre langage pour intégrer ces nouveautés.

Concernant la potentielle et future phase d'implémentation, un point important a été souligné : l'importance que ce soit le système qui contrôle les bonnes combinaisons de requêtes et non l'utilisateur. C'est en effet capital pour garder une bonne interprétation du langage. De plus comme le souligne notre méthodologie, un même système est souvent utilisé par des utilisateurs dont les niveaux d'expertise sont différents.

Bibliographie

[1] https ://docs.oracle.com/javase/8/docs/api/java/lang/thread.html, (Date de l'accès 6 Mai 2016).

[2] https ://eclipse.org/aspectj/, (Date de l'accès 1 Avril 2016).

[3] https ://neo4j.com/, (Date de l'accès 6 Mai 2016).

[4] http ://soft.vub.ac.be/soul/, (Date de l'accès 15 Decembre 2015).

[5] https ://www.geogebra.org/, (Date de l'accès 15 Avril 2016).

[6] http ://tyruba.sourceforge.net/, (Date de l'accès 20 Octobre 2015).

[7] http ://www.cs.ubc.ca/labs/spl/projects/jquery/, (Date de l'accès 10 Octobre 2015).

[8] Tiago L Alves, Jurriaan Hage, and Peter Rademaker. A comparative study of code query technologies. In *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, pages 145–154. IEEE, 2011.

[9] Anil Chawla and Alessandro Orso. A generic instrumentation framework for collecting dynamic information. *ACM SIGSOFT Software Engineering Notes*, 29(5) :1–4, 2004.

[10] Neil Coffey. http ://www.javamex.com/tutorials/memory/instrumentation.shtml, (Date de l'accès 3 Février 2016) 2011.

[11] Brian De Alwis and Gail C Murphy. Answering conceptual queries with ferret. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 21–30. IEEE, 2008.

[12] Coen De Roover, Carlos Noguera, Andy Kellens, and Vivane Jonckers. The soul tool suite for querying programs in symbiosis with eclipse. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 71–80. ACM, 2011.

[13] Kris De Volder. Jquery : A generic code browser with a declarative configuration language. In *Practical Aspects of Declarative Languages*, pages 88–102. Springer, 2006.

[14] Mickaël Delahaye. Instrumentation de code java. 2007.

[15] Judith E. Grass. Object-oriented design archaeology with cia++. *Computing Systems*, 5(1) :5–67, 1992.

[16] Elnar Hajiyev. *CodeQuest : Source Code Querying with Datalog*. PhD thesis, Citeseer, 2005.

[17] Elnar Hajiyev, Mathieu Verbaere, and Oege De Moor. Codequest : Scalable source code queries with datalog. In *ECOOP 2006–Object-Oriented Programming*, pages 2–27. Springer, 2006.

[18] Elnar Hajiyev, Mathieu Verbaere, Oege de Moor, and Kris De Volder. Codequest : querying source code with datalog. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 102–103. ACM, 2005.

[19] Natheer K. Gharaibeh Islah K. Gharaibeh. Towards arabic noun phrase extractor (anpe) using information retrieval techniques. *Science and academic publishing*, 2012.

[20] Jamel Eddine Jridi. Formulation interactive des requêtes pour l'analyse et la compréhension du code source. 2011.

[21] Markus Kimmig, Martin Monperrus, and Mira Mezini. Querying source code with natural language. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 376–379. IEEE Computer Society, 2011.

[22] Mark A. Linton. *Queries and Views of Programs Using a Relational Database System*. PhD thesis, EECS Department, University of California, Berkeley, Dec 1983. OMEGA.

[23] É. Gaussier M.-R. Amini. *Recherche d'Information - Applications, modèles et algorithmes [archive]*. Eyrolles, 2013.

[24] Vania Marangozova-Martin and Generoso Pagano. Gestion de traces d'exécution pour le systèmes embarqués : contenu et stockage. 2014.

[25] Mike McGavin, Tim Wright, and Stuart Marshall. Visualisations of execution traces (vet) : an interactive plugin-based visualisation tool. In *Proceedings of the 7th Australasian User interface conference-Volume 50*, pages 153–160. Australian Computer Society, Inc., 2006.

[26] Philip Meas. Visualizations pour la compréhension de programme avec un outil metacase. 2014.

[27] Santanu Paul and Atul Prakash. Querying source code using an algebraic query language. In *Software Maintenance, 1994. Proceedings., International Conference on*, pages 127–136. IEEE, 1994.

[28] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 13–22. IEEE, 1999.

[29] Bruce Tognazzini. First principles of interaction design. *Interaction design solutions for the real world, AskTog*, 2003.

[30] Raoul-Gabriel Urma and Alan Mycroft. Programming language evolution via source code query languages. In *Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools*, pages 35–38. ACM, 2012.

[31] Raoul-Gabriel Urma and Alan Mycroft. Source-code queries with graph databases—with application to programming language usage and evolution. *Science of Computer Programming*, 97 :127–134, 2015.

[32] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. *DSL engineering : Designing, implementing and using domain-specific languages*. dslbook. org, 2013.

Annexe

A. ANNEXE A

Attribute	Value type
Ev. Type	{ method-call, constructor-call, init-block-call, thread-call, read-field, write-field, throw exception, catch exception, undefined }
Ev. Type -List	Sequence of Ev. Type
Prev.Ban.Ev.Type-List	Sequence of Ev. Type
Elem. Type	{package, class, interface, init-block, method, constructor, field}
Elem. Type-List	Sequence of Elem.Type
Name	String of letters, digits or special caracters
Var-Name	String of Maj letters and digits
Var-Name-List	Sequence of Var-Name

Nonterminals	Synthesized Attributes	Inherited Attributes
<event type>		Type Prev.Ban.Ev.Type-List
<event type list>		Ev. Type-List Prev.Ban.Ev.Type-List
<right arrow>		Ev. Type-List
<left arrow>		Ev. Type-List
<left event>		Ev. Type-List
<right event>		Ev. Type-List Prev.Ban.Ev.Type-List
<double event>		Ev. Type-List
<event param>		Ev. Type-List
<event param variable>		Ev. Type-List
<event param trans clos>		Ev. Type-List
<char>		Name
<maj char>		Name
<number>		Name

<number p>		Name
<digit>		Name
<no zero digit>		Name
<elem identifier>		Name
<elem identifier char>		Name
<variable>		Var-Name
<variable p>		Var-Name
<variable char>		Var-Name
<path info>		Var-Name
<event>		Prev.Ban.Ev.Type-List

| étant utilisé par la grammaire BNF, nous écrirons '|' pour représenter le caractère |

QUERY		
<query>	::=	<match clause> <from clause> <where clause> <group by clause> <print clause>;
MATCH		
<match clause>	::=	MATCH <model sequence> |ϵ
<model sequence>	::=	<model> |<model> & <model sequence>
<model>	::=	'|' <path expr> '|' <event> '|' <path expr> '|' Prev.Ban.Ev.Type-List(<model>) <- Prev.Ban.Ev.Type-List(<event>) |'|' <path expr> '|' <event> <model>$_2$ Prev.Ban.Ev.Type-List(<model>) <- Prev.Ban.Ev.Type-List(<event>) **Condition:** **Foreach** ' Ev. Type' **of** Ev. Type-List(<event>) If 'Ev.Type' is member of Prev.Ban.Ev.Type-List(<model>$_2$) Then error(" wrong event ") |<path expr> Prev.Ban.Ev.Type-List(<model>) <- {}
<event>	::=	<right event> Prev.Ban.Ev.Type-List(<event>) <- Prev.Ban.Ev.Type-List(<right event>) | <double event> Prev.Ban.Ev.Type-List(<event>) <- Prev.Ban.Ev.Type-List(<double event>)
<double event>	::=	<left event> <right event> Prev.Ban.Ev.Type-List(<double event>) <- Prev.Ban.Ev.Type-List(<right event>) **Condition:** **Foreach** ' Ev. Type' **of** Ev. Type-List (<right event>) If 'Ev.Type' is member of Prev.Ban.Ev.Type-List(<left event>) Then error(" wrong event ")

<right event>	::=	: <event type list><event param> <right arrow> Prev.Ban.Ev.Type-List(**<right event>**) **<-** Prev.Ban.Ev.Type-List(<event type list>) **If empty(*Ev. Type-List*(<event type list>))** **Then** ***Ev. Type-List***(**<right event>**) **<-** ***Ev. Type-List***(<right arrow>) **Else** ***Ev. Type-List***(**<right event>**) **<-** ***Ev. Type-List***(**<event type list>**) ***Condition :*** **Foreach** 'Ev. Type' **of** ***Ev. Type-List***(<event type list>) ***If*** ('Ev. Type') ***is not a member of Ev. Type-List***(<right arrow>) Then error ("Not permitted event type selectionned") ***If*** ('Ev. Type') ***is a member of Ev. Type-List***(<event param>) Then error ("Not permitted transitive closure parameter for types selectionned ")
<left event>	::=	<left arrow><event param> <event type list> : Prev.Ban.Ev.Type-List (<left event>)**<-** {read-field, write-field, throw-exception, catch-exception, thread-call} **If empty(*Ev. Type-List*(<event type list>))** **Then** ***Ev. Type-List***(**<left event>**) **<-** ***Ev. Type-List***(<left arrow>) **Else** ***Ev. Type-List***(**<left event>**) **<-** ***Ev. Type-List***(**<event type list>**) ***Condition :*** **Foreach** 'Ev. Type' **of** ***Ev. Type-List***(<event type list>) ***If*** ('Ev. Type') ***is not a member of Ev. Type-List***(<left arrow>) Then error ("Not permitted event type selectionned") ***If*** ('Ev. Type') ***is a member of Ev. Type-List***(<event param>) Then error ("Not permitted transitive closure parameter for types selectionned ")
<event type list>	::=	**ϵ** ***Ev. Type-List***(**<event type list>**) **<-** ***{}*** Prev.Ban.Ev.Type-List(**<event type list>**)) <- {read-field, write-field, throw-exception, catch-exception} | <event type>, <event type list>$_2$ ***Ev. Type-List***(**<event type list>**) **<-** ***cons(Ev. Type*** (**<event type>**)***, Ev. Type-List***(**<event type list>**$_2$))

		Prev.Ban.Ev.Type-List(**<event type list>**)) <- ***union-list***(Prev.Ban.Ev.Type-List(**<event type>**)), Prev.Ban.Ev.Type-List ***(<event type list>2))***
<event type>	:=	c ***Ev. Type*** **(<event type>)** ***<- constructor-call*** Prev.Ban.Ev.Type-List(**<event type>**)) <- {read-field, write-field, throw-exception, catch-exception} | m ***Ev. Type*** **(<event type>)** ***<- method-call*** Prev.Ban.Ev.Type-List(**<event type>**)) <- {read-field, write-field, throw-exception, catch-exception} | t ***Ev. Type*** **(<event type>)** ***<- thread-call*** Prev.Ban.Ev.Type-List(**<event type>**)) <- {read-field, write-field, throw-exception, catch-exception} | b ***Ev. Type*** **(<event type>)** ***<- init-block-call*** Prev.Ban.Ev.Type-List(**<event type>**)) <- {throw-exception, catch-exception} | r ***Ev. Type*** **(<event type>)** ***<- read-field*** Prev.Ban.Ev.Type-List(**<event type>**)) <- {read-field, write-field, throw-exception, catch-exception} | w ***Ev. Type*** **(<event type>)** ***<- write-field*** Prev.Ban.Ev.Type-List(**<event type>**)) <- {read-field, write-field, throw-exception, catch-exception} | x ***Ev. Type*** **(<event type>)** ***<- throw-exception*** Prev.Ban.Ev.Type-List(**<event type>**)) <- {read-field, write-field, throw-exception, catch-exception} | /x\ ***Ev. Type*** **(<event type>)** ***<- catch-exception*** Prev.Ban.Ev.Type-List(**<event type>**)) <- {read-field, write-field, throw-exception, catch-exception}
<right arrow>	:=	-->***Ev. Type-List(<right arrow>) <- {method-call, constructor-call}*** | ~~>

		Ev. Type-List (<right arrow>) <-{init-block-call, thread-call} | ~~> *Ev. Type-List (<right arrow>) <- {method-call, constructor-call, init-block-call, thread-call }* | >> *Ev. Type-List (<right arrow>) <- {read-field, write-field, throw-exception, catch exception}* | -->> *Ev. Type-List (<right arrow>) <- {method call, constructor call, read-field, write-field, throw-exception, catch exception }* | ~~>> *Ev. Type-List (<right arrow>) <- { init-block-call, thread-call, read-field, write-field, throw-exception, catch-exception }* |~~>> *Ev. Type-List (<right arrow>) <- { init-block-call, thread-call, method-call, constructor-call, read-field, write-field, throw-exception, catch-exception }*
<left arrow>	:=	<-- *Ev. Type-List (<right arrow>) <-{method-call, constructor-call}*
<event param>	:=	{<event param variable><event param trans clos>} *Ev. Type-List(* **<event param>)** *<-* *Ev. Type-List(* **<event param trans clos>)** |ϵ *Ev. Type-List(* **<event param>)** *<-* {}
<event param variable>	:=	<event variable> |ϵ
<event param trans clos>	:=	(<event param trans clos value>) *Ev. Type-List(* **<event param trans clos>)** *<- { thread-call, read-field, write-field, throw-exception, catch-exception }* |ϵ *Ev. Type-List (<event param trans clos>) <- { }*
<event param trans clos value>	:=	<comp operator> number |<comp operator> number , <comp operator> number

		| *
<comp operator>	:=	> | < | =
FROM		
<from clause>	:=	FROM SCENARIO <elem identifier> | ϵ
WHERE		
<where clause>	:=	WHERE <where content> | ϵ
<where content>	:=	<where content> & <where content p> | <where content p>
<where content p>	:=	<elem type assign.p> | <path assign p.> | <signature assign. p> | <inheritance model p> | <event time comp. p> | <dyn. type. assign. p> | <dyn. type. comp. p> | <set integration p>
<set integration p>	:=	<set integration> | (<set integration> OR <set integration p>)
<elem type assign.p>	:=	<elem type assign> | (<elem type assign> OR <elem type assign.p>)
<path assign. p>	:=	<path assign> | (<path assign> OR <path assign p.>)
<signature assign. p>	:=	<signature assign.> | (<signature assign.> OR <signature assign. p>
<inheritance model p>	:=	<inheritance model> | (<inheritance model> OR <inheritance model. p>)
<event time comp. p>	:=	<event time comp.> | (<event time comp.> OR <event time comp.p>)

<dyn. type. assign. p>	::=	<dyn. type. assign.> |(<dyn. type. assign.> OR <dyn. type. assign.p>)
<dyn. type. comp. p>	::=	<dyn. type. comp.> |(<dyn. type. comp.> OR <dyn. type. comp.p>)
<var list>	::=	ϵ |<all variable> , <var list>
<set integration>	::=	{<var list>} <not>WITHIN(<query>)
<elem type assign.>	::=	<elem predicat> (<variable>)
<elem predicat>	::=	ISPACAKGE |ISCLASS |ISINTERFACE |ISBLOCK |ISCONSTRUCTOR |ISMETHOD |ISFIELD
<path assign.>	::=	<variable> = <path expr> |<variable> = <path expr> <static signature p>
<signature assign.>	::=	<variable> = <static signature> |<event variable> = <dyn. signature>
<static signature>	::=	'<elem identifier>' <static signature p>
<static signature p>	::=	| ϵ |{<accessibility> <static> <property>} |{\<static>\} | {< accessibility><static><property><synchronized><native> (<params type>) >> <return type>} | {<accessibility> (<params type>) \} | {<accessibility><static><volatile > <transient ><java type>}
<params type>	::=	ϵ |<param type p> |<param type p>, <params type>
<param type p>	::=	<param property> <java type>

		| ..
<java type>	::=	char | byte | short | int | long | float | double | boolean | String | <path elem idendifier> | *
<return type>	::=	:<java type> | void
<dyn. signature>	::=	{(<params value>) >> <literal>}} {(<params value>) \ } | {OLD. <literal> NEW. <literal>} | {<literal>}
<params value>	::=	| * | <param value p> | <param value p>, <params value>
<param value p>	::=	<literal> | ..
<accessibility>	::=	<not>public, <not>protected, <not>'friendly', <not>private
<volatile >	::=	<not>volatile | ϵ
<transient >	::=	<not>transient | ϵ
<synchronized>	::=	<not>synchronized | ϵ
<native>	::=	<not>native | ϵ
<static>	::=	<not>static | ϵ
<property>	::=	<not>abstract | <not> final | ϵ
<param property>	::=	<not>final | ϵ
<literal>	::=	<numeral> | <floating-point literal> | <boolean literal> | <character literal> | <string literal>

		| <null literal> |:'<path elem idendifier>' |*
<string literal>	::=	<string character> |<string character><string literal>
<floating-point literal>	::=	<numeral>.<zero numeral>
<boolean literal>	::=	true |false
<null literal>	::=	null
<path elem idendifier>	::=	<elem identifier> |<elem identifier> . <path elem identifier>
<event time comp.>	::=	<event variable> <event time comp.op> <event variable> |<event variable> <event time comp. op> <event time comp.>
<event time comp. op>	::=	< |>
<inheritance model>	::=	'|' <variable>'|' <inheritance relation> '|'<variable>'|' |'|' <variable>'|' <inheritance relation> <inheritance model>
<inheritance relation>	::=	≪ <inheritance type> > |≪ <inheritance type> or <inheritance type> >
<inheritance type>	::=	i |e
<dyn. type. assign.>	::=	ISINSTANCE (<variable>)
<dyn. type. comp.>	::=	instanceOf(<event variable>) <dyn. type. comp. op> instanceOf(<event variable>) |threadOf(<event variable>) <dyn. type. comp. op> threadOf(<event variable>)
<dyn. type. comp. op>	::=	<not>= |=
PRINT		
<print clause>	::=	PRINT <print sequence> |ϵ
<print sequence>	::=	|<print content>, <print sequence>

<print content>	::=	<all variable> |<function>(<all variable>)
GROUP BY		
<group by clause>	::=	GROUP BY <variable sequence> |ϵ
<variable sequence>	::=	<all variable>, <variable sequence>
Used by clauses		
<function>	::=	PackageView |ClassView |MethodView |<not>Req |Distinct ...
<path expr>	::=	<path info> *Var-Name-List(<path expr>) <- cons(Var-Name(*<path info>*), empty-list)* |<path info><path sep><path expr>$_2$ *Var-Name-List(<path expr>) <- cons(Var-Name(*<path info>*), Var-Name-List(<path expr>$_2$))* ***Condition:*** ***If Var-Name(Var-Name(***<path info>***)) is not a member of Var-Name-List(<path expr>$_2$)*** ***Then error(" ")*** ***Else error(" Duplicate variable in 'Path expression' ")***
<path sep>	::=	. |..
<path info>	::=	<variable> Var-***Name*** *(*<path info>*)* **<-** ***Var-Name*** *(*<variable>*)* |'<elem identifier>' Var-***Name*** *(*<path info>*)* **<-** '' |* Var-***Name*** *(*<path info>*)* **<-** ''
<all variable>	::=	<event variable>

		|<variable>
<event variable>	:=	@ <variable>
<variable>	:=	<maj char> <variable p> **Var-Name (<variable>) <- str-concat(*Name*(<maj char>), Var-Name (<variable p>))**
<variable p>	:=	ϵ **Var-Name (<variable p>) <- ''** |<variable char> <variable p>$_2$ **Var-Name (<variable p>) <- str-concat (Var-*Name*(<variable char>), Var-*Name*(<variable p>$_2$))**
<variable char>	:=	<maj char> **Var-Name (<variable char>) <- *Name*(<maj char>)** |<digit> **Var-Name (<variable char>) <- *Name*(<digit>)**
<elem identifier>	:=	<elem identifier char> ***Name*(<elem identifier>) <- *Name*(<elem identifier char>)** |<elem identifier char> <elem identifier>$_2$ ***Name*(<elem identifier>) <- str-concat(*Name*(<elem identifier char>), *Name*(<elem identifier>)$_2$)**
<elem identifier char>	:=	$ ***Name*(<elem identifier char>) <- '$'** | _ ***Name*(<elem identifier char>) <- '_'** | <char> ***Name*(<elem identifier char>) <- *Name*(<char>)** | <digit> ***Name*(<elem identifier char>) <- '*Name*(<digit>)** | * ***Name*(<elem identifier char>) <- '* '**
<not>	:=	ϵ |!

<char>	::=	<maj char> ***Name***(<char>) **<-** ***Name***(<maj char>) | [a-z] ***Name***(<char>) **<-** ***'a'*** ... ***Name***(<char>) **<-** ***'z'***
<string character>	::=	[Unicode input character except " and \]
<maj char>	::=	[A-Z] ***Name***(<maj char>) **<-** ***'A'*** .... ***Name***(<maj char>) **<-** ***'Z'***
<number>	::=	<digit> ***Name***(<number>) **<-** ***Name***(<digit>) |<no zero digit><number>$_2$ ***Name***(<number>) <- str-concat(***Name***(<no zero digit>), ***Name***(<number p>))
<zero numeral>	::=	<digit> |<digit><zero numeral>
<number p>	::=	<digit> ***Name***(<number p>) **<-** ***Name***(<digit>) | digit><number p>$_2$ ***Name***(<number p>) <- str-concat(***Name***(<no zero digit>), ***Name***(<number p>$_2$))
<digit>	::=	[0-9] ***Name***(<digit>) **<-** ***'0'*** .... ***Name***(<digit>) **<-** ***'9'***
<no zero digit>	::=	[1-9] ***Name***(<no zero digit>)**<-** ***'1'*** .... ***Name***(<no zero digit>)**<-** ***'9'***

Auxiliary Functions		
str-concat('String1','String2')	=	Returns the concatenation of 'String1' followed by 'String2'
cons(first, rest)	=	[first | rest]
empty([])	=	true
union-list(list1, list2)	=	If empty(list1) Then return list2 Else foereach 'Ev.Type' of list1 If 'Ev.Type' is not member of list2 Then cons ('Ev. Type', list2) return list2

Grammaire abstraite

<query>	:=	<match clause><from clause><where clause><group by clause><print clause>

<match clause>	:=	**MATCH** <model>$^{+}$ |**ϵ**
<model>	:=	<path expr> <event> <model> | <path expr>
<event>	:=	<right event> |<double event>
<double event>	:=	<left event><right event>
<right event>	:=	: <event type>*<event param><right arrow>
<left event>	:=	<left arrow><event param><event type>* :
<event type>	:=	c | m | t | b | r | w | x | /x\
<right arrow>	:=	--> | ~~> | ~~> | >> | -->> | ~~>> | ~~>>
<left arrow>	:=	<--
<event param>	:=	{<event param variable> <event param trans clos>} |ϵ
<event param variable>	:=	<event variable> |ϵ
<event param trans clos>	:=	(<event param trans clos value>) | ϵ
<event param trans clos value>	:=	<comp operator>number | <comp operator> number <comp operator>number | *
<comp operator>	:=	> | < | =

<from clause>	:=	**FROM** <elem identifier> |ϵ

<where clause>	:=	**WHERE** <where content p>$^{+}$ |ϵ
<where content p>	:=	<path assign.>$^{+}$ |<elem type assign.>$^{+}$ |<signature assign.>$^{+}$ |<inheritance model>$^{+}$ |<event time comp.>$^{+}$ |<dyn. type. assign.>$^{+}$ |<dyn. type. comp.>$^{+}$ |<set integration>$^{+}$
<elem type assign.>	:=	<elem predicat><variable>
<elem predicat>	:=	ISPACAKGE | ISCLASS | ISINTERFACE | ISBLOCK | ISCONSTRUCTOR | ISMETHOD | ISFIELD
<path assign.>	:=	<variable> = <path expr>

		|<variable> = <path expr> <static signature p>
<signature assign.>	::=	<variable> = <static signature> |<event variable> = <dyn. signature>
<static signature>	::=	'<elem identifier>'<static signature p>
<static signature p>	::=	| ϵ
		|{<accessibility> <static><property>}
		|{\<static>\}
		|{< accessibility><static><property><synchronized><native> (<param type p>*) <return type>}
		|{<accessibility> (<params type p>*) \}
		| {<accessibility><static> <volatile > <transient ><java type>}
<param type p>	::=	<java type> |..
<dyn. signature>	::=	{(<params value p>*)<literal>}}
		{(<params value p>*)\}
		|{OLD. <literal> NEW. <literal>}
		|{<literal>}
<param value p>	::=	<param property><literal> |..
<accessibility>	::=	<not>public, <not>protected, <not>'friendly', <not>private
<volatile >	::=	<not>volatile |ϵ
<transient >	::=	<not>transient |ϵ
<synchronized>	::=	<not>synchronized|ϵ
<native>	::=	<not>native |ϵ
<static>	::=	<not>static |ϵ
<property>	::=	<not>abstract |<not> final |ϵ
<param property>	::=	<not>final| ϵ
<java type>	::=	char |byte |short |int |long |float |double |boolean |String |<path elem idendifier> |*
<return type>	::=	<java type> | void
<literal>	::=	<numeral> | <floating-point literal> | <boolean literal> | <character literal> | <string literal>
		| <null literal> |:'<path elem idendifier>' |*
<string literal>	::=	<string character> |<string character><string literal>
<floating-point literal>	::=	<numeral>.<zero numeral>
<boolean literal>	::=	true |false

<null literal>	::=	null
<path elem idendifier>	::=	<elem identifier> | <elem identifier> . <path elem identifier>
<set integration>	::=	{<all var>$^{+}$}<not>WITHIN(<query>)
<event time comp.>	::=	<event variable> <event time comp. op> <event variable> | <event variable> <event time comp. op> <event time comp.>
<event time comp. op>	::=	< | >
<inheritance model>	::=	<variable><inheritance relation><variable> | <variable> <inheritance relation><inheritance model>
<inheritance relation>	::=	≪ <inheritance type> > | ≪ <inheritance type> or <inheritance type> >
<inheritance type>	::=	i | e
<dyn. type. assign.>	::=	ISINSTANCE <event variable>
<dyn. type. comp.>	::=	instanceOf <event variable><dyn. type. comp. op> instanceOf <event variable> | threadOf <event variable> <dyn. type. comp. op> threadOf <event variable>
<dyn. type. comp. op>	::=	!= | =

<print clause>	::=	**PRINT** <print sequence> | ϵ
<print sequence>	::=	<print content>, <print sequence>
<print content>	::=	<all variable> | <function>(<all variable>)

<group by clause>	::=	**GROUP BY** <variable sequence> | ϵ
<variable sequence>	::=	<all variable>, <variable sequence>

<function>	::=	PackageView | ClassView | MethodView | <not>Req | Distinct | ...
<path expr>	::=	<path info> | <path info><path sep> <path expr>
<path sep>	::=	. | ..
<not>	::=	! | ϵ
<path info>	::=	<variable> | ' < elem identifier> ' | *
<all variable>	::=	<event variable> | <variable>

<event variable>	::=	@<variable>
<variable>	::=	<maj char> <variable p>
<variable p>	::=	ϵ |<variable char> <variable p>
<variable char>	::=	<maj char> |<digit>
<elem identifier>	::=	<elem identifier char> | <elem identifier char><elem identifier>
<elem identifier char>	::=	$ | _ | <char> | <digit> | *
<char>	::=	<maj char> | [a-z]
<string character>	::=	[Unicode input character except " and \]
<maj char>	::=	[A-Z]
<number>	::=	<digit> |<no zero digit><number>
<zero numeral>	::=	<digit>|<digit><zero numeral>
<number p>	::=	<digit> | digit><number p>
<digit>	::=	[0-9]
<no zero digit>	::=	[1-9]

B. ANNEXE B

Domaine sémantique:

Ac.modifier = {public, protected, friendly, private}

St.modifier = {static, notstatic}

Sy.modifier ={synchronized, notsynchronized}

Na.modifier = {native, notnative}

Vo.modifier = {volatile, notvolatile}

Tr.modifier ={transient, nottransient}

Af.modifier ={abstractNotFinal, finalNotAbstract, notAbstractNotFinal}

Sf.modifier = { strictfp, notStrictfp}

De.modifier = { default, notDefault}

Param.modifier = {final, notFinal}

"Nous assumons qu'il existe une fonction sémantique implicite qui lie chaque <elem identifier> et <path elem idendifier> du monde syntaxique à une valeur du monde sémantique. Ces valeurs sont les <elem identifier> et <path elem idendifier> eux même de tel sorte que ***id [[EI]] =EI*** *et* ***id [[PEI]] =PEI****."*

JavaTypeSet = {char, byte, short, int, long, float, double, boolean, String} ∪ <path elem idendifier>

ParamStore = { $\mathbb{N}^*$ ⟶ **Param.modifier x JavaTypeSet** } application où $\mathbb{N}^*$ représente la position du paramètre dans la signature d'une méthode/constructeur

---

PackageSet = <elem identifier>

PackagePathSet = **PackageSet***

ClassSet = **AbstractNotStrictFpClassSet** ∪ **AbstractStrictFpClassSet** ∪ **FinalNotAbstractNotStrictFpClassSet** ∪ **FinalNotAbstractStrictFpClassSet** ∪ **NotFinalNotAbstractNotStrictFpClassSet** ∪ **NotFinalNotAbstractStrictFpClassSet**

NotFinalClassSet = **AbstractNotStrictFpClassSet** ∪**AbstractStrictFpClassSet** ∪ **NotFinalNotAbstractNotStrictFpClassSet** ∪ **NotFinalNotAbstractStrictFpClassSet**

ClassPathSet = **AbstractNotStrictFpClassPathSet** ∪ **AbstractStrictFpClassPathSet** ∪ **FinalNotAbstractNotStrictFpClassPathSet** ∪ **FinalNotAbstractStrictFpClassPathSet** ∪ **NotFinalNotAbstractNotStrictFpClassPathSet** ∪ **NotFinalNotAbstractStrictFpClassPathSet**

NotFinalClassPathSet = **AbstractNotStrictFpClassPathSet** ∪ **AbstractStrictFpClassPathSet** ∪ **NotFinalNotAbstractNotStrictFpClassPathSet** ∪ **NotFinalNotAbstractStrictFpClassPathSet**

NotAbstractClassPathSet = **FinalNotAbstractNotStrictFpClassPathSet** ∪ **FinalNotAbstractStrictFpClassPathSet** ∪ **NotFinalNotAbstractNotStrictFpClassPathSet** ∪ **NotFinalNotAbstractStrictFpClassPathSet**

1-AbstractNotStrictFpClassSet = ClassModifierSet x <elem identifier>

- **Ac.modifierRest = Ac.modifier** \ {protected, friendly, private}
- **Af.modifierRest = Af.modifier** \{finalNotAbstract, notAbstractNotFinal}
- **Sf.modifierRest = Sf.modifier** \{strictfp}
- **ClassModifierSet = Ac.modifierRest x Af.modifierRest x Sf.modifierRest**

AbstractNotStrictFpClassPathSet = PackagePathSet x AbstractNotStrictFpClassSet

2-AbstractStrictFpClassSet = ClassModifierSet x <elem identifier>

- **Ac.modifierRest = Ac.modifier** \ {protected, friendly, private}
- **Af.modifierRest = Af.modifier** \{finalNotAbstract, notAbstractNotFinal}
- **Sf.modifierRest = Sf.modifier** \{notStrictfp}
- **ClassModifierSet = Ac.modifierRest x St.modifier x Af.modifierRest x Sf.modifierRest**

AbstractStrictFpClassPathSet = PackagePathSet x AbstractStrictFpClassSet

3-NotFinalNotAbstractNotStrictFpClassSet = ClassModifierSet x <elem identifier>

- **Ac.modifierRest = Ac.modifier** \ {protected, friendly, private}
- **Af.modifierRest = Af.modifier** \{abstractNotFinal, finalNotAbstract}
- **Sf.modifierRest = Sf.modifier** \{strictfp}
- **ClassModifierSet = Ac.modifierRest x St.modifier x Af.modifierRest x Sf.modifierRest**

NotFinalNotAbstractNotStrictFpClassPathSet = PackagePathSet x NotFinalNotAbstractNotStrictFpClassSet

4-FinalNotAbstractNotStrictFpClassSet = ClassModifierSet x <elem identifier>

- **Ac.modifierRest = Ac.modifier** \ {protected, friendly, private}
- **Af.modifierRest = Af.modifier** \{abstractNotFinal, notAbstractNotFinal}
- **Sf.modifierRest = Sf.modifier** \{strictfp}
- **ClassModifierSet = Ac.modifierRest x St.modifier x Af.modifierRest x Sf.modifierRest**

FinalNotAbstractNotStrictFpClassPathSet = PackagePathSet x FinalNotAbstractNotStrictFpClassSet

5-NotFinalNotAbstractStrictFpClassSet = ClassModifierSet x <elem identifier>

- **Ac.modifierRest = Ac.modifier** \ {protected, friendly, private}
- **Af.modifierRest = Af.modifier** \{abstractNotFinal, finalNotAbstract}
- **Sf.modifierRest = Sf.modifier** \{notStrictfp}
- **ClassModifierSet = Ac.modifierRest x St.modifier x Af.modifierRest x Sf.modifierRest**

NotFinalNotAbstractStrictFpClassPathSet = PackagePathSet x NotFinalNotAbstractStrictFpClassSet

6-FinalNotAbstractStrictFpClassSet = ClassModifierSet x <elem identifier>

- **Ac.modifierRest = Ac.modifier** \ {protected, friendly, private}
- **Af.modifierRest = Af.modifier** \{abstractNotFinal, notAbstractNotFinal}
- **Sf.modifierRest = Sf.modifier** \{notStrictfp}
- **ClassModifierSet = Ac.modifierRest x St.modifier x Af.modifierRest x Sf.modifierRest**

FinalNotAbstractStrictFpClassPathSet = PackagePathSet x FinalNotAbstractStrictFpClassSet

---

InnerClassSet = ClassInnerClassSet ∪ InterfaceInnerClassSet ∪ BlockInnerClassSet

InnerClassPathSet = ClassInnerClassPathSet ∪InterfaceInnerClassPathSet ∪ BlockInnerClassPathSet

InnerNotAbstractClassPathSet = ClassFinalNotAbstractNotStrictFpInnerClassPathSet ∪ ClassFinalNotAbstractStrictFpInnerClassPathSet ∪ ClassNotFinalNotAbstractNotStrictFpInnerClassPathSet ∪ ClassNotFinalNotAbstractStrictFpInnerClassPathSet ∪ InterfaceFinalNotAbstractNotStrictFpInnerClassPathSet ∪ InterfaceFinalNotAbstractStrictFpInnerClassPathSet ∪ InterfaceNotFinalNotAbstractNotStrictFpInnerClassPathSet ∪ InterfaceNotFinalNotAbstractStrictFpInnerClassPathSet

NotFinalInnerClassSet = ClassAbstractNotStrictFpInnerClassSet ∪ ClassAbstractStrictFpInnerClassSet ∪ ClassNotFinalNotAbstractNotStrictFpInnerClassSet ∪ ClassNotFinalNotAbstractStrictFpInnerClassSet ∪ InterfaceAbstractNotStrictFpInnerClassSet ∪ InterfaceAbstractStrictFpInnerClassSet ∪ InterfaceNotFinalNotAbstractNotStrictFpInnerClassSet ∪ InterfaceNotFinalNotAbstractStrictFpInnerClassSet ∪ BlockAbstractNotStrictFpInnerClassSet ∪ BlockAbstractStrictFpInnerClassSet ∪ BlockNotFinalNotAbstractNotStrictFpInnerClassSet ∪ BlockNotFinalNotAbstractStrictFpInnerClassSet

NotFinalInnerClassPathSet = ClassAbstractNotStrictFpInnerClassPathSet ∪ ClassAbstractStrictFpInnerClassPathSet ∪
ClassNotFinalNotAbstractNotStrictFpInnerClassPathSet ∪ ClassNotFinalNotAbstractStrictFpInnerClassPathSet ∪
InterfaceAbstractNotStrictFpInnerClassPathSet ∪ InterfaceAbstractStrictFpInnerClassPathSet ∪
InterfaceNotFinalNotAbstractNotStrictFpInnerClassPathSet ∪ InterfaceNotFinalNotAbstractStrictFpInnerClassPathSet ∪
BlockAbstractNotStrictFpInnerClassPathSet ∪ BlockAbstractStrictFpInnerClassPathSet ∪
BlockNotFinalNotAbstractNotStrictFpInnerClassPathSet ∪ BlockNotFinalNotAbstractStrictFpInnerClassPathSet

ClassInnerClassSet = ClassAbstractNotStrictFpInnerClassSet ∪ ClassAbstractStrictFpInnerClassSet ∪
ClassFinalNotAbstractNotStrictFpInnerClassSet ∪ ClassFinalNotAbstractStrictFpInnerClassSet ∪
ClassNotFinalNotAbstractNotStrictFpInnerClassSet ∪ ClassNotFinalNotAbstractStrictFpInnerClassSet

InterfaceInnerClassSet = InterfaceAbstractNotStrictFpInnerClassSet ∪ InterfaceAbstractStrictFpInnerClassSet ∪
InterfaceFinalNotAbstractNotStrictFpInnerClassSet ∪ InterfaceFinalNotAbstractStrictFpInnerClassSet ∪
InterfaceNotFinalNotAbstractNotStrictFpInnerClassSet ∪ InterfaceNotFinalNotAbstractStrictFpInnerClassSet

BlockInnerClassSet = BlockAbstractNotStrictFpInnerClassSet ∪ BlockAbstractStrictFpInnerClassSet ∪
BlockFinalNotAbstractNotStrictFpInnerClassSet ∪ BlockFinalNotAbstractStrictFpInnerClassSet∪
BlockNotFinalNotAbstractNotStrictFpInnerClassSet ∪ BlockNotFinalNotAbstractStrictFpInnerClassSet

ClassInnerClassPathSet = ClassAbstractNotStrictFpInnerClassPathSet ∪ ClassAbstractStrictFpInnerClassPathSet ∪
ClassFinalNotAbstractNotStrictFpInnerClassPathSet ∪ ClassFinalNotAbstractStrictFpInnerClassPathSet ∪
ClassNotFinalNotAbstractNotStrictFpInnerClassPathSet ∪ ClassNotFinalNotAbstractStrictFpInnerClassPathSet

InterfaceInnerClassPathSet = InterfaceAbstractNotStrictFpInnerClassPathSet ∪ InterfaceAbstractStrictFpInnerClassPathSet ∪
InterfaceFinalNotAbstractNotStrictFpInnerClassPathSet ∪ InterfaceFinalNotAbstractStrictFpInnerClassPathSet ∪
InterfaceNotFinalNotAbstractNotStrictFpInnerClassPathSet ∪ InterfaceNotFinalNotAbstractStrictFpInnerClassPathSet

BlockInnerClassPathSet = BlockAbstractNotStrictFpInnerClassPathSet ∪ BlockAbstractStrictFpInnerClassPathSet ∪
BlockFinalNotAbstractNotStrictFpInnerClassPathSet ∪ BlockFinalNotAbstractStrictFpInnerClassPathSet ∪
BlockNotFinalNotAbstractNotStrictFpInnerClassPathSet ∪ BlockNotFinalNotAbstractStrictFpInnerClassPathSet.

ClassInnerClassPath = {ClassPathSet ∪ (ClassPathSet x ClassInnerClassSet*), InterfaceInnerClassPathSet ∪ (InterfaceInnerClassPathSet x
ClassInnerClassSet*), BlockInnerClassPathSet ∪ (BlockInnerClassPathSet x ClassInnerClassSet*)}

InterfaceInnerClassPath = InterfacePathSet ∪ InterfaceInnerInterfacePath ∪ ClassInnerInterfacePath

BlockInnerClassPath = BlockPathSet ∪ ConstructorPathSet ∪ MethodPathSet

1-ClassAbstractNotStrictFpInnerClassSet = ClassModifierSet x <elem identifier>

- **Af.modifierRest = Af.modifier** \{finalNotAbstract, notAbstractNotFinal}
- **Sf.modifierRest = Sf.modifier** \{strictfp}
- **ClassModifierSet = Ac.modifier x St.modifier x Af.modifierRest x Sf.modifierRest**

ClassAbstractNotStrictFpInnerClassPathSet = ClassInnerClassPath x ClassAbstractNotStrictFpInnerClassSet

2-InterfaceAbstractNotStrictFpInnerClassSet = ClassModifierSet x <elem identifier>

- **Ac.modifierRest = Ac.modifier** \{protected, friendly, private}
- **Af.modifierRest = Af.modifier** \{finalNotAbstract, notAbstractNotFinal}
- **Sf.modifierRest = Sf.modifier** \{strictfp}
- **ClassModifierSet = Ac.modifierRest x St.modifier x Af.modifierRest x Sf.modifierRest**

InterfaceAbstractNotStrictFpInnerClassPathSet = InterfaceInnerClassPath x InterfaceAbstractNotStrictFpInnerClassSet

3-BlockAbstractNotStrictFpInnerClassSet = ClassModifierSet x <elem identifier>

- **Af.modifierRest = Af.modifier** \{finalNotAbstract, notAbstractNotFinal}
- **Sf.modifierRest = Sf.modifier** \{strictfp}
- **ClassModifierSet = St.modifier x Af.modifierRest x Sf.modifierRest**

BlockAbstractNotStrictFpInnerClassPathSet = BlockInnerClassPath x BlockAbstractNotStrictFpInnerClassSet

4-ClassAbstractStrictFpInnerClassSet = ClassModifierSet x <elem identifier>

- **Af.modifierRest = Af.modifier** \{finalNotAbstract, notAbstractNotFinal}
- **Sf.modifierRest = Sf.modifier** \{notStrictfp}
- **ClassModifierSet = Ac.modifier x St.modifier x Af.modifierRest x Sf.modifierRest**

ClassAbstractStrictFpInnerClassPathSet = ClassInnerClassPath x ClassAbstractStrictFpInnerClassSet

5-InterfaceAbstractStrictFpInnerClassSet = ClassModifierSet x <elem identifier>

- **Ac.modifierRest = Ac.modifier** \{protected, friendly, private}
- **Af.modifierRest = Af.modifier** \{finalNotAbstract, notAbstractNotFinal}
- **Sf.modifierRest = Sf.modifier** \{notStrictfp}
- **ClassModifierSet = Ac.modifierRest x St.modifier x Af.modifierRest x Sf.modifierRest**

InterfaceAbstractStrictFpInnerClassPathSet = InterfaceInnerClassPath x InterfaceAbstractStrictFpInnerClassSet

6-BlockAbstractStrictFpInnerClassSet = ClassModifierSet x <elem identifier>

- **Af.modifierRest = Af.modifier** \{finalNotAbstract, notAbstractNotFinal}
- **Sf.modifierRest = Sf.modifier** \{notStrictfp}
- **ClassModifierSet = St.modifier x Af.modifierRest x Sf.modifierRest**

BlockAbstractStrictFpInnerClassPathSet = BlockInnerClassPath x BlockAbstractStrictFpInnerClassSet

7-ClassFinalNotAbstractNotStrictFpInnerClassSet = ClassModifierSet x <elem identifier>

- **Af.modifierRest = Af.modifier** \{abstractNotFinal, notAbstractNotFinal}
- **Sf.modifierRest = Sf.modifier** \{strictfp}
- **ClassModifierSet = Ac.modifier x St.modifier x Af.modifierRest x Sf.modifierRest**

ClassFinalNotAbstractNotStrictFpInnerClassPathSet = ClassInnerClassPath x ClassFinalNotAbstractNotStrictFpInnerClassSet

8-ClassNotFinalNotAbstractNotStrictFpInnerClassSet = ClassModifierSet x <elem identifier>

- **Af.modifierRest = Af.modifier** \{abstractNotFinal, finalNotAbstract}
- **Sf.modifierRest = Sf.modifier** \{strictfp}
- **ClassModifierSet = Ac.modifier x St.modifier x Af.modifierRest x Sf.modifierRest**

ClassNotFinalNotAbstractNotStrictFpInnerClassPathSet = ClassInnerClassPath x ClassNotFinalNotAbstractNotStrictFpInnerClassSet

9-InterfaceFinalNotAbstractNotStrictFpInnerClassSet = ClassModifierSet x <elem identifier>

- **Ac.modifierRest = Ac.modifier** \{protected, friendly, private}
- **Af.modifierRest = Af.modifier** \{abstractNotFinal, notAbstractNotFinal}
- **Sf.modifierRest = Sf.modifier** \{strictfp}
- **ClassModifierSet = Ac.modifierRest x St.modifier x Af.modifierRest x Sf.modifierRest**

InterfaceFinalNotAbstractNotStrictFpInnerClassPathSet = InterfaceInnerClassPath x InterfaceFinalNotAbstractNotStrictFpInnerClassSet

10-InterfaceNotFinalNotAbstractNotStrictFpInnerClassSet = ClassModifierSet x <elem identifier>

- **Ac.modifierRest = Ac.modifier** \{protected, friendly, private}
- **Af.modifierRest = Af.modifier** \{abstractNotFinal, finalNotAbstract}
- **Sf.modifierRest = Sf.modifier** \{strictfp}
- **ClassModifierSet = Ac.modifierRest x St.modifier x Af.modifierRest x Sf.modifierRest**

InterfaceNotFinalNotAbstractNotStrictFpInnerClassPathSet = InterfaceInnerClassPath x InterfaceNotFinalNotAbstractNotStrictFpInnerClassSet

11-BlockFinalNotAbstractNotStrictFpInnerClassSet = ClassModifierSet x <elem identifier>

- **Af.modifierRest = Af.modifier** \{abstractNotFinal, notAbstractNotFinal}
- **Sf.modifierRest = Sf.modifier** \{strictfp}
- **ClassModifierSet = St.modifier x Af.modifierRest x Sf.modifierRest**

BlockFinalNotAbstractNotStrictFpInnerClassPathSet = BlockInnerClassPath x BlockFinalNotAbstractNotStrictFpInnerClassSet

12-BlockNotFinalNotAbstractNotStrictFpInnerClassSet = ClassModifierSet x <elem identifier>

- **Af.modifierRest = Af.modifier** \{abstractNotFinal, finalNotAbstract}
- **Sf.modifierRest = Sf.modifier** \{strictfp}
- **ClassModifierSet = St.modifier x Af.modifierRest x Sf.modifierRest**

BlockNotFinalNotAbstractNotStrictFpInnerClassPathSet = BlockInnerClassPath x BlockNotFinalNotAbstractNotStrictFpInnerClassSet

13-ClassFinalNotAbstractStrictFpInnerClassSet = ClassModifierSet x <elem identifier>

- **Af.modifierRest = Af.modifier** \{abstractNotFinal, notAbstractNotFinal}
- **Sf.modifierRest = Sf.modifier** \{notStrictfp}
- **ClassModifierSet = Ac.modifier x St.modifier x Af.modifierRest x Sf.modifierRest**

ClassFinalNotAbstractStrictFpInnerClassPathSet = ClassInnerClassPath x ClassFinalNotAbstractStrictFpInnerClassSet

14-ClassNotFinalNotAbstractStrictFpInnerClassSet = ClassModifierSet x <elem identifier>

- **Af.modifierRest = Af.modifier** \{abstractNotFinal, finalNotAbstract }
- **Sf.modifierRest = Sf.modifier** \{notStrictfp}
- **ClassModifierSet = Ac.modifier x St.modifier x Af.modifierRest x Sf.modifierRest**

ClassNotFinalNotAbstractStrictFpInnerClassPathSet = ClassInnerClassPath x ClassNotFinalNotAbstractStrictFpInnerClassSet

15-InterfaceFinalNotAbstractStrictFpInnerClassSet = ClassModifierSet x <elem identifier>

- **Ac.modifierRest = Ac.modifier** \{protected, friendly, private}
- **Af.modifierRest = Af.modifier** \ { abstractNotFinal, notAbstractNotFinal }
- **Sf.modifierRest = Sf.modifier** \{notStrictfp}
- **ClassModifierSet = Ac.modifier x St.modifier x Af.modifierRest x Sf.modifierRest**

InterfaceFinalNotAbstractStrictFpInnerClassPathSet = InterfaceInnerClassPath x InterfaceFinalNotAbstractStrictFpInnerClassSet

16-InterfaceNotFinalNotAbstractStrictFpInnerClassSet = ClassModifierSet x <elem identifier>

- **Ac.modifierRest = Ac.modifier** \{protected, friendly, private}
- **Af.modifierRest = Af.modifier** \{abstractNotFinal, finalNotAbstract }
- **Sf.modifierRest = Sf.modifier** \{notStrictfp}
- **ClassModifierSet = Ac.modifier x St.modifier x Af.modifierRest x Sf.modifierRest**

InterfaceNotFinalNotAbstractStrictFpInnerClassPathSet = InterfaceInnerClassPath x InterfaceNotFinalNotAbstractStrictFpInnerClassSet

17-BlockFinalNotAbstractStrictFpInnerClassSet= ClassModifierSet x <elem identifier>

- **Af.modifierRest = Af.modifier** \{abstractNotFinal, notAbstractNotFinal}
- **Sf.modifierRest = Sf.modifier** \{notStrictfp}
- **ClassModifierSet = St.modifier x Af.modifierRest x Sf.modifierRest**

BlockFinalNotAbstractStrictFpInnerClassPathSet = BlockInnerClassPath x BlockFinalNotAbstractStrictFpInnerClassSet

18-BlockNotFinalNotAbstractStrictFpInnerClassSet = ClassModifierSet x <elem identifier>

- **Af.modifierRest = Af.modifier** \{abstractNotFinal, finalNotAbstract }
- **Sf.modifierRest = Sf.modifier** \{notStrictfp}
- **ClassModifierSet = St.modifier x Af.modifierRest x Sf.modifierRest**

BlockNotFinalNotAbstractStrictFpInnerClassPathSet = BlockInnerClassPath x BlockNotFinalNotAbstractStrictFpInnerClassSet

---

InterfaceSet = StrictFpInterfaceSet ∪ NotStrictFpInterfaceSet

IntefacePathSet = StrictIntefacePathSet ∪ NotStrictFpIntefacePathSet

1-StrictFpInterfaceSet = InterfaceModifierSet x <elem identifier>

- **Ac.modifierRest = Ac.modifier** \ {protected, friendly, private}
- **InterfaceAf.modifier.restrict = Af.modifier** \ {finalNotAbstract}
- **Sf.modifierRest = Sf.modifier** \ {notStrictfp}
- **InterfaceModifierSet = Ac.modifier x St.modifier x InterfaceAf.modifier.restrict x Sf.modifierRest**

StrictFpIntefacePathSet = PackagePathSet x StrictFpInterfaceSet

2-NotStrictFpInterfaceSet = InterfaceModifierSet x <elem identifier>

- **Ac.modifierRest = Ac.modifier** \ {protected, friendly, private}
- **InterfaceAf.modifier.restrict = Af.modifier** \ {finalNotAbstract}
- **Sf.modifierRest = Sf.modifier** \ {strictfp}
- **InterfaceModifierSet = Ac.modifier x St.modifier x InterfaceAf.modifier.restrict x Sf.modifierRest**

NotStrictFpIntefacePathSet = PackagePathSet x NotStrictFpInterfaceSet

---

InnerInterfaceSet = InterfaceStrictfpInnerInterfaceSet ∪ ClassStrictfpInnerInterfaceSet ∪ InterfaceNotStrictfpInnerInterfaceSet ∪ ClassNotStrictfpInnerInterfaceSet

InnerIntefacePathSet = InterfaceStrictfpInnerInterfacePathSet ∪ ClassStrictfpInnerInterfacePathSet ∪ InterfaceNotStrictfpInnerInterfacePathSet ∪ ClassNotStrictfpInnerInterfacePathSet

InterfaceInnerInterfaceSet = InterfaceStrictfpInnerInterfaceSet ∪ InterfaceNotStrictfpInnerInterfaceSet

InterfaceInnerInterfacePath = {InterfacePathSet ⊕ (InterfacePathSet x InterfaceInnerInterfaceSet*), InterfaceInnerClassPathSet ⊕ (InterfaceInnerClassPathSet x InterfaceInnerInterfaceSet *)}

ClassInnerInterfacePath = ClassPathSet ∪ ClassInnerClassPath ∪ InterfaceInnerClassPath ∪ BlockInnerClassPath

1-InterfaceStrictfpInnerInterfaceSet = InnerInterfaceModifierSet x <elem identifier>

- **Ac.modifierRest = Ac.modifier** \ {protected, friendly, private}
- **InterfaceAf.modifier.restrict = Af.modifier** \ {finalNotAbstract}
- **Sf.modifierRest = Sf.modifier** \ {notStrictfp}
- **InnerInterfaceModifierSet = Ac.modifierRest x InterfaceAf.modifier.restrict x Sf.modifier**

InterfaceStrictfpInnerInterfacePathSet = InterfaceInnerInterfacePath x InterfaceStrictfpInnerInterfaceSet

2-ClassStrictfpInnerInterfaceSet = InnerInterfaceModifierSet x <elem identifier>

- **InterfaceAf.modifier.restrict = Af.modifier** \{finalNotAbstract}
- **Sf.modifierRest = Sf.modifier** \{notStrictfp}
- **InnerInterfaceModifierSet = Ac.modifier x InterfaceAf.modifier.restrict x Sf.modifier**

ClassStrictfpInnerInterfacePathSet = ClassInnerInterfacePath x ClassStrictfpInnerInterfaceSet

3-InterfaceNotStrictfpInnerInterfaceSet = InnerInterfaceModifierSet x <elem identifier>

- **Ac.modifierRest = Ac.modifier** \ {protected, friendly, private}
- **InterfaceAf.modifier.restrict = Af.modifier** \{finalNotAbstract}
- **Sf.modifierRest = Sf.modifier** \{strictfp}
- **InnerInterfaceModifierSet = Ac.modifierRest x InterfaceAf.modifier.restrict x Sf.modifier**

InterfaceNotStrictfpInnerInterfacePathSet = InterfaceInnerInterfacePath x InterfaceNotStrictfpInnerInterfaceSet

4-ClassNotStrictfpInnerInterfaceSet = InnerInterfaceModifierSet x <elem identifier>

- **InterfaceAf.modifier.restrict = Af.modifier** \{finalNotAbstract}
- **Sf.modifierRest = Sf.modifier** \{strctfp}
- **InnerInterfaceModifierSet = Ac.modifier x InterfaceAf.modifier.restrict x Sf.modifier**

ClassNotStrictfpInnerInterfacePathSet = ClassInnerInterfacePath x ClassNotStrictfpInnerInterfaceSet

---

BlockSet = StaticBlockSet ∪ StaticBlockSet

BlockPathSet = StaticBlockPathSet ∪ NotStaticBlockPathSet

1-StaticBlockSet = St.modifierRest x $\mathbb{N}^+$

- **St.modifierRest = St.modifier \ {notStatic}**

StaticBlockPathSet = BlockPath x StaticBlockSet

- **BlockPath = ClassPathSet ∪ InnerClassPathSet**

2-NotStaticBlockSet = St.modifierRest x $\mathbb{N}^+$

- **St.modifierRest = St.modifier \ {static}**

NotStaticBlockPathSet = BlockPath x NotStaticBlockSet

- **BlockPath = ClassPathSet ∪ InnerClassPathSet**

1-ConstructorSet = Ac.modifier x ParamStore x <elem identifier>

ConstructorPathSet = ConstructorPath x ConstructorSet

- **ConstructorPath = ClassPathSet ∪ InnerClassPathSet**

MethodSet = AbstractNotStrictFpClassFinalStaticMethodSet ∪ AbstractNotStrictFpClassFinalNotStaticMethodSet ∪ AbstractNotStrictFpClassNotFinalAbstractMethodSet ∪ AbstractNotStrictFpClassNotFinalNotAbstractStaticMethodSet ∪ AbstractNotStrictFpClassNotFinalNotAbstractNotStaticMethodSet ∪ FinalNotAbstractNotStrictFpClassStaticMethodSet ∪ FinalNotAbstractNotStrictFpClassNotStaticMethodSet ∪ NotFinalNotAbstractNotStrictFpClassFinalStaticMethodSet ∪ NotFinalNotAbstractNotStrictFpClassFinalNotStaticMethodSet ∪ NotFinalNotAbstractNotStrictFpClassNotFinalStaticMethodSet ∪ NotFinalNotAbstractNotStrictFpClassNotFinalNotStaticMethodSet ∪ AbstractStrictFpClassFinalStaticMethodSet ∪ AbstractStrictFpClassFinalNotSaticMethodSet ∪ AbstractStrictFpClassNotFinalAbstractMethodSet ∪ AbstractStrictFpClassNotFinalNotAbstractStaticMethodSet ∪ AbstractStrictFpClassNotFinalNotAbstractNotStaticMethodSet ∪ FinalNotAbstractStrictFpClassStaticMethodSet ∪ FinalNotAbstractStrictFpClassNotStaticMethodSet ∪ NotFinalNotAbstractStrictFpClassFinalStaticMethodSet ∪ NotFinalNotAbstractStrictFpClassFinalNotStaticMethodSet ∪ NotFinalNotAbstractStrictFpClassNotFinalStaticMethodSet ∪ NotFinalNotAbstractStrictFpClassNotFinalNotStaticMethodSet ∪ StrictFpInterfaceAbstractMethodSet ∪ NotStrictFpInterfaceAbstractMethodSet ∪ StrictFpInterfaceNotAbstractStaticMethodSet ∪

StrictFpInterfaceNotAbstractNotStaticMethodSet ∪ NotStrictFpInterfaceNotAbstractSaticMethodSet ∪
NotStrictFpInterfaceNotAbstractNotStaticMethodSet

MethodPathSet = AbstractNotStrictFpClassFinalStaticMethodPathSet ∪ AbstractNotStrictFpClassFinalNotStaticMethodPathSet ∪
AbstractNotStrictFpClassNotFinalAbstractMethodPathSet ∪ AbstractNotStrictFpClassNotFinalNotAbstractStaticMethodPathSet ∪
AbstractNotStrictFpClassNotFinalNotAbstractNotStaticMethodPathSet ∪ FinalNotAbstractNotStrictFpClassStaticMethodPathSet ∪
FinalNotAbstractNotStrictFpClassNotStaticMethodPathSet ∪ NotFinalNotAbstractNotStrictFpClassFinalStaticMethodPathSet ∪
NotFinalNotAbstractNotStrictFpClassFinalNotStaticMethodPathSet ∪ NotFinalNotAbstractNotStrictFpClassNotFinalStaticMethodPathSet
∪ NotFinalNotAbstractNotStrictFpClassNotFinalNotStaticMethodPathSet ∪ AbstractStrictFpClassFinalStaticMethodPathSet ∪
AbstractStrictFpClassFinalNotStaticMethodPathSet ∪ AbstractStrictFpClassNotFinalAbstractMethodPathSet ∪
AbstractStrictFpClassNotFinalNotAbstractStaticMethodPathSet ∪ AbstractStrictFpClassNotFinalNotAbstractNotStaticMethodPathSet ∪
FinalNotAbstractStrictFpClassStaticMethodPathSet ∪ FinalNotAbstractStrictFpClassNotStaticMethodPathSet∪
NotFinalNotAbstractStrictFpClassFinalStaticMethodPathSet ∪ NotFinalNotAbstractStrictFpClassFinalNotStaticMethodPathSet ∪
NotFinalNotAbstractStrictFpClassNotFinalStaticMethodPathSet ∪ NotFinalNotAbstractStrictFpClassNotFinalNotStaticMethodPathSet ∪
StrictFpInterfaceAbstractMethodPathSet ∪ NotStrictFpInterfaceAbstractMethodPathSet ∪
StrictFpInterfaceNotAbstractStaticMethodPathSet ∪ StrictFpInterfaceNotAbstractNotStaticMethodPathSet ∪
NotStrictFpInterfaceNotAbstractStaticMethodPathSet ∪ NotStrictFpInterfaceNotAbstractNotStaticMethodPathSet

AbstractMethodePathSet = AbstractNotStrictFpClassNotFinalAbstractMethodPathSet ∪
AbstractStrictFpClassNotFinalAbstractMethodPathSet ∪ StrictFpInterfaceAbstractMethodPathSet ∪
NotStrictFpInterfaceAbstractMethodPathSet

NotAbstractMethodePathSet = NotAbstractNotStaticMethodPathSet ∪ NotAbstractStaticMethodPathSet

NotAbstractNotStaticMethodPathSet = AbstractNotStrictFpClassFinalNotStaticMethodPathSet ∪
AbstractNotStrictFpClassNotFinalNotAbstractNotStaticMethodPathSet ∪ FinalNotAbstractNotStrictFpClassNotStaticMethodPathSet ∪
NotFinalNotAbstractNotStrictFpClassFinalNotStaticMethodPathSet ∪
NotFinalNotAbstractNotStrictFpClassNotFinalNotStaticMethodPathSet ∪ AbstractStrictFpClassFinalNotStaticMethodPathSet ∪
AbstractStrictFpClassNotFinalNotAbstractNotStaticMethodPathSet ∪ FinalNotAbstractStrictFpClassNotStaticMethodPathSet ∪
NotFinalNotAbstractStrictFpClassFinalNotStaticMethodPathSet ∪ NotFinalNotAbstractStrictFpClassNotFinalNotStaticMethodPathSet ∪
StrictFpInterfaceNotAbstractNotStaticMethodPathSet ∪ NotStrictFpInterfaceNotAbstractNotStaticMethodPathSet

NotAbstractStaticMethodPathSet = AbstractNotStrictFpClassFinalStaticMethodPathSet ∪
AbstractNotStrictFpClassNotFinalNotAbstractStaticMethodPathSet ∪ FinalNotAbstractNotStrictFpClassStaticMethodPathSet ∪
NotFinalNotAbstractNotStrictFpClassFinalStaticMethodPathSet ∪ NotFinalNotAbstractNotStrictFpClassNotFinalStaticMethodPathSet ∪
NotFinalNotAbstractStrictFpClassNotFinalStaticMethodPathSet ∪ AbstractStrictFpClassNotFinalNotAbstractStaticMethodPathSet ∪
FinalNotAbstractStrictFpClassStaticMethodPathSet ∪ NotFinalNotAbstractStrictFpClassFinalStaticMethodPathSet ∪
StrictFpInterfaceNotAbstractStaticMethodPathSet ∪ NotStrictFpInterfaceNotAbstractStaticMethodPathSet

FinalMethodPathSet = AbstractNotStrictFpClassFinalNotStaticMethodPathSet ∪
FinalNotAbstractNotStrictFpClassNotStaticMethodPathSet ∪ FinalNotAbstractStrictFpClassNotStaticMethodPathSet ∪
AbstractStrictFpClassFinalNotStaticMethodPathSet ∪ FinalNotAbstractStrictFpClassNotStaticMethodPathSet ∪
NotFinalNotAbstractStrictFpClassFinalNotStaticMethodPathSet ∪ AbstractNotStrictFpClassFinalStaticMethodPathSet ∪
FinalNotAbstractNotStrictFpClassStaticMethodPathSet ∪ NotFinalNotAbstractNotStrictFpClassFinalStaticMethodPathSet ∪
FinalNotAbstractStrictFpClassStaticMethodPathSet ∪ NotFinalNotAbstractStrictFpClassFinalStaticMethodPathSet

NotFinalMethodPathSet = AbstractNotStrictFpClassNotFinalNotAbstractNotStaticMethodPathSet ∪ NotFinalNotAbstractNotStrictFpClassNotFinalNotStaticMethodPathSet ∪ AbstractStrictFpClassFinalNotStaticMethodPathSet AbstractStrictFpClassNotFinalNotAbstractNotStaticMethodPathSet ∪ NotFinalNotAbstractStrictFpClassNotFinalNotStaticMethodPathSet ∪ AbstractNotStrictFpClassNotFinalNotAbstractStaticMethodPathSet ∪ NotFinalNotAbstractNotStrictFpClassNotFinalStaticMethodPathSet ∪ NotFinalNotAbstractStrictFpClassNotFinalStaticMethodPathSet ∪ AbstractStrictFpClassNotFinalNotAbstractStaticMethodPathSet.

1- AbstractNotStrictFpClassFinalStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> **x JavaTypeSet**

- **NaSfModifierRest = (Na.modifier x Sf.modifier) \ {**(native, strictfp)**}**
- **Af.modifierRest = Af.modifier** \ { abstractNotFinal, notAbstractNotFinal }
- **St.modifierRest = St.modifier** \ {notStatic}
- **MethodModifierSet = Ac.modifier x St.modifierRest x Af.modifierRest x Sy.modifier x NaSfModifierRest**

AbstractNotStrictFpClassFinalStaticMethodPathSet = ANSCP x AbstractNotStrictFpClassFinalStaticMethodSet

- **ANSCP = AbstractNotStrictFpClassPathSet ∪ ClassAbstractNotStrictFpInnerClassPathSet ∪ InterfaceAbstractNotStrictFpInnerClassPathSet ∪ BlockAbstractNotStrictFpInnerClassPathSet**

2- AbstractNotStrictFpClassFinalNotSaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> **x JavaTypeSet**

- **NaSfModifierRest = (Na.modifier x Sf.modifier) \ {**(native, strictfp)**}**
- **Af.modifierRest = Af.modifier** \ { abstractNotFinal, notAbstractNotFinal }
- **St.modifierRest = St.modifier** \ {static}
- **MethodModifierSet = Ac.modifier x St.modifierRest x Af.modifierRest x Sy.modifier x NaSfModifierRest**

AbstractNotStrictFpClassFinalNotStaticMethodPathSet = ANSCP x AbstractNotStrictFpClassFinalNotStaticMethodSet

- **ANSCP = AbstractNotStrictFpClassPathSet ∪ ClassAbstractNotStrictFpInnerClassPathSet ∪ InterfaceAbstractNotStrictFpInnerClassPathSet ∪ BlockAbstractNotStrictFpInnerClassPathSet**

Remarque : Nous pouvons avoir des méthodes 'StrictFp' avec un classe 'NotStrictfp '

3- AbstractNotStrictFpClassNotFinalAbstractMethodSet = MethodModifierSet x ParamStore x <elem identifier> x JavaTypeSet

- **NaSfModifierRest = (Na.modifier x Sf.modifier) \ {(**native, strictfp**)}**
- **baseAf.modifierRest = Af.modifier** \ { finalNotAbstract, notFinalNotAbstract }
- **BaseModifierSet = Ac.modifier x St.modifier x baseAf.modifierRest x Sy.modifier x NaSfModifierRest**
- **Af.modifierRest = Af.modifier** \ { finalNotAbstract, notAbstractNotFinal }
- **ModifierRestOperand1 = Ac.modifier x St.modifier x Af.modifierRest x Sy.modifier x Na.modifier x Sf.modifier**
- **ModifierRestOperand2 = {(public,** notstatic , abstractNotFinal, notsynchronized, notnative, notStrictfp**),**
  (protected, notstatic , abstractNotFinal, notsynchronized, notnative, notStrictfp**) }**
  "Représente les combinaisons possibles des 'modifier' lorsque le 'modifier' Abstract est présent"
- **MethodModifierRestrict1 = ModifierRestOperand1 \ ModifierRestOperand2**
- **MethodModifierSet = BaseModifierSet \ MethodModifierRestrict1**

AbstractNotStrictFpClassNotFinalAbstractMethodPathSet = ANSCP x AbstractNotStrictFpClassNotFinalAbstractMethodSet

- **ANSCP = AbstractNotStrictFpClassPathSet ∪ ClassAbstractNotStrictFpInnerClassPathSet ∪ InterfaceAbstractNotStrictFpInnerClassPathSet ∪ BlockAbstractNotStrictFpInnerClassPathSet**

4- AbstractNotStrictFpClassNotFinalNotAbstractStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> **x JavaTypeSet**

- **NaSfModifierRest = (Na.modifier x Sf.modifier) \ {**(native, strictfp)**}**
- **Af.modifierRest = Af.modifier** \ { abstractNotFinal, finalNotAbstract }
- **St.modifierRest = St.modifier** \ {notStatic}
- **MethodModifierSet = Ac.modifier x St.modifierRest x Af.modifierRest x Sy.modifier x NaSfModifierRest**

AbstractNotStrictFpClassNotFinalNotAbstractStaticMethodPathSet = ANSCP x AbstractNotStrictFpClassNotFinalNotAbstractStaticMethodSet

- ANSCP = AbstractNotStrictFpClassPathSet ∪ ClassAbstractNotStrictFpInnerClassPathSet ∪ InterfaceAbstractNotStrictFpInnerClassPathSet ∪ BlockAbstractNotStrictFpInnerClassPathSet

5- AbstractNotStrictFpClassNotFinalNotAbstractNotStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> **x JavaTypeSet**

- **NaSfModifierRest = (Na.modifier x Sf.modifier) \ {**(native, strictfp)**}**
- **Af.modifierRest = Af.modifier** \ { abstractNotFinal, finalNotAbstract }
- **St.modifierRest = St.modifier** \ {static}
- **MethodModifierSet = Ac.modifier x St.modifierRest x Af.modifierRest x Sy.modifier x NaSfModifierRest**

AbstractNotStrictFpClassNotFinalNotAbstractNotStaticMethodPathSet = ANSCP x AbstractNotStrictFpClassNotFinalNotAbstractNotStaticMethodSet

- ANSCP = AbstractNotStrictFpClassPathSet ∪ ClassAbstractNotStrictFpInnerClassPathSet ∪ InterfaceAbstractNotStrictFpInnerClassPathSet ∪ BlockAbstractNotStrictFpInnerClassPathSet

6 -FinalNotAbstractNotStrictFpClassStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> **x JavaTypeSet**

- **NaSfModifierRest = (Na.modifier x Sf.modifier) \ {**(native, strictfp)**}**
- **Af.modifierRest = Af.modifier \ {**notFinalNotAbstract, abstractNotFinal**}**
- **St.modifierRest = St.modifier \ {**notStatic**}**
- **MethodModifierSet = Ac.modifier x St.modifier x Af.modifierRest x Sy.modifier x NaSfModifierRest**

FinalNotAbstractNotStrictFpClassStaticMethodPathSet = NANSCP x FinalNotAbstractNotStrictFpClassStaticMethodSet

- **NANSCP = FinalNotAbstractNotStrictFpClassPathSet ∪ ClassFinalNotAbstractNotStrictFpInnerClassPathSet ∪ InterfaceFinalNotAbstractNotStrictFpInnerClassPathSet ∪ BlockFinalNotAbstractNotStrictFpInnerClassPathSet**

7- FinalNotAbstractNotStrictFpClassNotStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> **x JavaTypeSet**

- **NaSfModifierRest = (Na.modifier x Sf.modifier) \ {**(native, strictfp)**}**
- **Af.modifierRest = Af.modifier \ {**notFinalNotAbstract, abstractNotFinal**}**
- **St.modifierRest = St.modifier \ {**notStatic**}**
- **MethodModifierSet = Ac.modifier x St.modifierRest x Af.modifierRest x Sy.modifier x NaSfModifierRest**

FinalNotAbstractNotStrictFpClassNotStaticMethodPathSet = NANSCP x FinalNotAbstractNotStrictFpClassNotStaticMethodSet

- **NANSCP = FinalNotAbstractNotStrictFpClassPathSet ∪ ClassFinalNotAbstractNotStrictFpInnerClassPathSet ∪ InterfaceFinalNotAbstractNotStrictFpInnerClassPathSet ∪ BlockFinalNotAbstractNotStrictFpInnerClassPathSet**

8- NotFinalNotAbstractNotStrictFpClassFinalStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> **x JavaTypeSet**

- **NaSfModifierRest = (Na.modifier x Sf.modifier) \ {**(native, strictfp)**}**
- **Af.modifierRest = Af.modifier \ {**abstractNotFinal, noFinalNotAbstract**}**
- **St.modifierRest = St.modifier \ {**notStatic**}**
- **MethodModifierSet = Ac.modifier x St.modifierRest x Af.modifierRest x Sy.modifier x NaSfModifierRest**

NotFinalNotAbstractNotStrictFpClassFinalStaticMethodPathSet = NANSCP x NotFinalNotAbstractNotStrictFpClassFinalStaticMethodSet

- **NANSCP = NotFinalNotAbstractNotStrictFpClassPathSet ∪ ClassNotFinalNotAbstractNotStrictFpInnerClassPathSet ∪ InterfaceNotFinalNotAbstractNotStrictFpInnerClassPathSet ∪ BlockNotFinalNotAbstractNotStrictFpInnerClassPathSet**

9- NotFinalNotAbstractNotStrictFpClassFinalNotStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> **x JavaTypeSet**

- **NaSfModifierRest = (Na.modifier x Sf.modifier) \ {**(native, strictfp)**}**
- **Af.modifierRest = Af.modifier \ {**abstractNotFinal, noFinalNotAbstract**}**
- **St.modifierRest = St.modifier \ {**static**}**
- **MethodModifierSet = Ac.modifier x St.modifierRest x Af.modifierRest x Sy.modifier x NaSfModifierRest**

NotFinalNotAbstractNotStrictFpClassFinalNotStaticMethodPathSet = NANSCP x NotFinalNotAbstractNotStrictFpClassFinalNotSaticMethodSet

- **NANSCP = NotFinalNotAbstractNotStrictFpClassPathSet ∪ ClassNotFinalNotAbstractNotStrictFpInnerClassPathSet ∪ InterfaceNotFinalNotAbstractNotStrictFpInnerClassPathSet ∪ BlockNotFinalNotAbstractNotStrictFpInnerClassPathSet**

10- NotFinalNotAbstractNotStrictFpClassNotFinalStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> **x JavaTypeSet**

- **NaSfModifierRest = (Na.modifier x Sf.modifier) \ {**(native, strictfp)**}**
- **Af.modifierRest = Af.modifier \ {**abstractNotFinal, finalNotAbstract**}**
- **St.modifierRest = St.modifier \ {**notStatic**}**
- **MethodModifierSet = Ac.modifier x St.modifierRest x Af.modifierRest x Sy.modifier x NaSfModifierRest**

NotFinalNotAbstractNotStrictFpClassNotFinalStaticMethodPathSet= NANSCP x NotFinalNotAbstractNotStrictFpClassNotFinalStaticMethodSet

- **NANSCP = NotFinalNotAbstractNotStrictFpClassPathSet ∪ ClassNotFinalNotAbstractNotStrictFpInnerClassPathSet ∪ InterfaceNotFinalNotAbstractNotStrictFpInnerClassPathSet ∪ BlockNotFinalNotAbstractNotStrictFpInnerClassPathSet**

11- NotFinalNotAbstractNotStrictFpClassNotFinalNotStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> **x JavaTypeSet**

- **NaSfModifierRest = (Na.modifier x Sf.modifier) \ {**(native, strictfp)**}**
- **Af.modifierRest = Af.modifier \ {**abstractNotFinal, finalNotAbstract**}**
- **St.modifierRest = St.modifier \ {**static**}**
- **MethodModifierSet = Ac.modifier x St.modifierRest x Af.modifierRest x Sy.modifier x NaSfModifierRest**

NotFinalNotAbstractNotStrictFpClassNotFinalNotStaticMethodPathSet = NANSCP x NotFinalNotAbstractNotStrictFpClassNotFinalNotStaticMethodSet

- **NANSCP = NotFinalNotAbstractNotStrictFpClassPathSet ∪ ClassNotFinalNotAbstractNotStrictFpInnerClassPathSet ∪ InterfaceNotFinalNotAbstractNotStrictFpInnerClassPathSet ∪ BlockNotFinalNotAbstractNotStrictFpInnerClassPathSet**

12- AbstractStrictFpClassFinalStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> x JavaTypeSet

- **Sf.modifierRest = Sf.modifier \ {notStrictfp}**
- **Af.modifierRest = Af.modifier \ {abstractNotFinal, notAbstractNotFinal}**
- **NaSfModifierRest = (Na.modifier x Sf.modifierRest) \ {(native, strictfp)}**
- **St.modifierRest = St.modifier \ {notStatic}**
- **MethodModifierSet = Ac.modifier x St.modifierRest x Af.modifierRest x Sy.modifier x NaSfModifierRest**

AbstractStrictFpClassFinalStaticMethodPathSet = ASCP x AbstractStrictFpClassFinalStaticMethodSet

- **ASCP = AbstractStrictFpClassPathSet ∪ ClassAbstractStrictFpInnerClassPathSet ∪ InterfaceAbstractStrictFpInnerClassPathSet ∪ BlockAbstractStrictFpInnerClassPathSet**

13- AbstractStrictFpClassFinalNotStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> x JavaTypeSet

- **Sf.modifierRest = Sf.modifier \ {notStrictfp}**
- **Af.modifierRest = Af.modifier \ {abstractNotFinal, notAbstractNotFinal}**
- **NaSfModifierRest = (Na.modifier x Sf.modifierRest) \ {(native, strictfp)}**
- **St.modifierRest = St.modifier \ {static}**
- **MethodModifierSet = Ac.modifier x St.modifierRest x Af.modifierRest x Sy.modifier x NaSfModifierRest**

AbstractStrictFpClassFinalNotStaticMethodPathSet = ASCP x AbstractStrictFpClassFinalNotSaticMethodSet

- **ASCP = AbstractStrictFpClassPathSet ∪ ClassAbstractStrictFpInnerClassPathSet ∪ InterfaceAbstractStrictFpInnerClassPathSet ∪ BlockAbstractStrictFpInnerClassPathSet**

14- AbstractStrictFpClassNotFinalAbstractMethodSet = MethodModifierSet x ParamStore x <elem identifier> **x JavaTypeSet**

- **Sf.modifierRest = Sf.modifier \ {**notStrictfp**}**
- **NaSfModifierRest = (Na.modifier x Sf.modifierRest) \ {**(native, strictfp)**}**
- **BaseAf.modifierRest = Af.modifier** \ { finalNotAbstract, notAbstractNotFinal }
- **BaseModifierSet = Ac.modifier x St.modifier x BaseAf.modifierRest x Sy.modifier x NaSfModifierRest**
- **MethodModifierRestrict = {(public,** notstatic , abstractNotFinal, notsynchronized, notnative, notStrictfp**) ,**
  (protected, notstatic , abstractNotFinal, notsynchronized, notnative, notStrictfp**) }**
  "Représente les combinaisons possibles des 'modifier' lorsque le 'modifier' Abstract est présent"
- **MethodModifierSet = BaseModifierSet \ MethodModifierRestrict**

AbstractStrictFpClassNotFinalAbstractMethodPathSet = ASCP x AbstractStrictFpClassNotFinalAbstractMethodSet

- **ASCP = AbstractStrictFpClassPathSet ∪ ClassAbstractStrictFpInnerClassPathSet ∪ InterfaceAbstractStrictFpInnerClassPathSet ∪ BlockAbstractStrictFpInnerClassPathSet**

15- AbstractStrictFpClassNotFinalNotAbstractStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> **x JavaTypeSet**

- **Sf.modifierRest = Sf.modifier \ {**notStrictfp**}**
- **NaSfModifierRest = (Na.modifier x Sf.modifierRest) \ {**(native, strictfp)**}**
- **BaseAf.modifierRest = Af.modifier** \ { finalNotAbstract, abstractNotFinal }
- **St.modifierRest = St.modifier \ {**notStatic**}**
- **MethodModifierSet = Ac.modifier x St.modifierRest x BaseAf.modifierRest x Sy.modifier x NaSfModifierRest**

AbstractStrictFpClassNotFinalNotAbstractStaticMethodPathSet = ASCP x AbstractStrictFpClassNotFinalNotAbstractStaticMethodSet

- **ASCP = AbstractStrictFpClassPathSet ∪ ClassAbstractStrictFpInnerClassPathSet ∪ InterfaceAbstractStrictFpInnerClassPathSet ∪ BlockAbstractStrictFpInnerClassPathSet**

16- AbstractStrictFpClassNotFinalNotAbstractNotStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> **x JavaTypeSet**

- **Sf.modifierRest = Sf.modifier \ {**notStrictfp**}**
- **NaSfModifierRest = (Na.modifier x Sf.modifierRest) \ {(**native, strictfp**)}**
- **BaseAf.modifierRest = Af.modifier** \ { finalNotAbstract, abstractNotFinal }
- **St.modifierRest = St.modifier \ {**static**}**
- **MethodModifierSet = Ac.modifier x St.modifierRest x BaseAf.modifierRest x Sy.modifier x NaSfModifierRest**

AbstractStrictFpClassNotFinalNotAbstractNotStaticMethodPathSet= ASCP x AbstractStrictFpClassNotFinalNotAbstractNotStaticMethodSet

- **ASCP = AbstractStrictFpClassPathSet ∪ ClassAbstractStrictFpInnerClassPathSet ∪ InterfaceAbstractStrictFpInnerClassPathSet ∪ BlockAbstractStrictFpInnerClassPathSet**

17- FinalNotAbstractStrictFpClassStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> **x JavaTypeSet**

- **Sf.modifierRest = Sf.modifier \ {**notStrictfp**}**
- **NaSfModifierRest = (Na.modifier x Sf.modifierRest) \ {(**native, strictfp**)}**
- **Af.modifierRest = Af.modifier \ {**abstractNotFinal, notAbstractNotFinal**}**
- **St.modifierRest = St.modifier \ {**notStatic**}**
- **MethodModifierSet = Ac.modifier x St.modifierRest x Af.modifierRest x Sy.modifier x NaSfModifierRest**

FinalNotAbstractStrictFpClassStaticMethodPathSet = NASCP x FinalNotAbstractStrictFpClassStaticMethodSet

- **NASCP = FinalNotAbstractStrictFpClassPathSet ∪ ClassFinalNotAbstractStrictFpInnerClassPathSet ∪ InterfaceFinalNotAbstractStrictFpInnerClassPathSet ∪ BlockFinalNotAbstractStrictFpInnerClassPathSet**

18- FinalNotAbstractStrictFpClassNotStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> **x JavaTypeSet**

- **Sf.modifierRest** = **Sf.modifier** \ {notStrictfp}
- **NaSfModifierRest** = **(Na.modifier x Sf.modifierRest)** \ {(native, strictfp)}
- **Af.modifierRest** = **Af.modifier** \ {abstractNotFinal, notAbstractNotFinal}
- **St.modifierRest** = **St.modifier** \ {static}
- **MethodModifierSet** = **Ac.modifier x St.modifierRest x Af.modifierRest x Sy.modifier x NaSfModifierRest**

FinalNotAbstractStrictFpClassNotStaticMethodPathSet = NASCP x FinalNotAbstractStrictFpClassNotStaticMethodSet

- **NASCP = FinalNotAbstractStrictFpClassPathSet ∪ ClassFinalNotAbstractStrictFpInnerClassPathSet ∪ InterfaceFinalNotAbstractStrictFpInnerClassPathSet ∪ BlockFinalNotAbstractStrictFpInnerClassPathSet**

19- NotFinalNotAbstractStrictFpClassFinalStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> **x JavaTypeSet**

- **Sf.modifierRest** = **Sf.modifier** \ {notStrictfp}
- **NaSfModifierRest** = **(Na.modifier x Sf.modifierRest)** \ {(native, strictfp)}
- **Af.modifierRest** = **Af.modifier** \ {abstractNotFinal, notAbstractNotFinal}
- **St.modifierRest** = **St.modifier** \ {notStatic}
- **MethodModifierSet** = **Ac.modifier x St.modifierRest x Af.modifierRest x Sy.modifier x NaSfModifierRest**

NotFinalNotAbstractStrictFpClassFinalStaticMethodPathSet = NASCP x NotFinalNotAbstractStrictFpClassFinalStaticMethodSet

- **NASCP = NotFinalNotAbstractStrictFpClassPathSet ∪ ClassNotFinalNotAbstractStrictFpInnerClassPathSet ∪ InterfaceNotfinalNotAbstractStrictFpInnerClassPathSet ∪ BlockNotFinalNotAbstractStrictFpInnerClassPathSet**

20- NotFinalNotAbstractStrictFpClassFinalNotStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> x JavaTypeSet

- **Sf.modifierRest = Sf.modifier \ {notStrictfp}**
- **NaSfModifierRest = (Na.modifier x Sf.modifierRest) \ {(native, strictfp)}**
- **Af.modifierRest = Af.modifier \ {abstractNotFinal, notAbstractNotFinal}**
- **St.modifierRest = St.modifier \ {static}**
- **MethodModifierSet = Ac.modifier x St.modifierRest x Af.modifierRest x Sy.modifier x NaSfModifierRest**

NotFinalNotAbstractStrictFpClassFinalNotStaticMethodPathSet = NASCP x NotFinalNotAbstractStrictFpClassFinalNotStaticMethodSet

- **NASCP = NotFinalNotAbstractStrictFpClassPathSet ∪ ClassNotFinalNotAbstractStrictFpInnerClassPathSet ∪ InterfaceNotfinalNotAbstractStrictFpInnerClassPathSet ∪ BlockNotFinalNotAbstractStrictFpInnerClassPathSet**

21- NotFinalNotAbstractStrictFpClassNotFinalStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> x JavaTypeSet

- **Sf.modifierRest = Sf.modifier \ {notStrictfp}**
- **NaSfModifierRest = (Na.modifier x Sf.modifierRest) \ {(native, strictfp)}**
- **Af.modifierRest = Af.modifier \ {abstractNotFinal, finalNotAbstract}**
- **St.modifierRest = St.modifier \ {notStatic}**
- **MethodModifierSet = Ac.modifier x St.modifierRest x Af.modifierRest x Sy.modifier x NaSfModifierRest**

NotFinalNotAbstractStrictFpClassNotFinalStaticMethodPathSet = NASCP x NotFinalNotAbstractStrictFpClassNotFinalStaticMethodSet

- **NASCP = NotFinalNotAbstractStrictFpClassPathSet ∪ ClassNotFinalNotAbstractStrictFpInnerClassPathSet ∪ InterfaceNotfinalNotAbstractStrictFpInnerClassPathSet ∪ BlockNotFinalNotAbstractStrictFpInnerClassPathSet**

22- NotFinalNotAbstractStrictFpClassNotFinalNotStaticMethodSet = MethodModifierSet x ParamStore x <elem identifier> x JavaTypeSet

- **Sf.modifierRest = Sf.modifier \ {notStrictfp}**
- **NaSfModifierRest = (Na.modifier x Sf.modifierRest) \ {(native, strictfp)}**
- **Af.modifierRest = Af.modifier \ {abstractNotFinal, finalNotAbstract}**
- **St.modifierRest = St.modifier \ {static}**
- **MethodModifierSet = Ac.modifier x St.modifierRest x Af.modifierRest x Sy.modifier x NaSfModifierRest**

NotFinalNotAbstractStrictFpClassNotFinalNotStaticMethodPathSet = NASCP x NotFinalNotAbstractStrictFpClassNotFinalNotStaticMethodSet

- **NASCP = NotFinalNotAbstractStrictFpClassPathSet ∪ ClassNotFinalNotAbstractStrictFpInnerClassPathSet ∪ InterfaceNotfinalNotAbstractStrictFpInnerClassPathSet ∪ BlockNotFinalNotAbstractStrictFpInnerClassPathSet**

Remarque: Attention il n'y pas de méthodes ('static' / 'default' avec 'final') dans une interface java 8 (contrairement aux classes abstraites).

23- StrictFpInterfaceAbstractMethodSet = InterfaceMethodModifier x ParamStore x <elem identifier> x JavaTypeSet

- **Ac.modifierMethInterface = Ac.modifier \ {protected, friendly, private}**
- **AbStDeModifier = {abstractNotStaticNotdefault}**
- **SfModifierRest = SfModifier \ {strictfp}**
- **InterfaceMethodModifier = Ac.modifierMethInterface x AbStDeModifier x SfModifierRest**

StrictFpInterfaceAbstractMethodPathSet = SIP x StrictFpInterfaceAbstractMethodSet

- **SIP = ClassStrictfpInnerInterfacePathSet ∪ InterfaceStrictfpInnerInterfacePathSet ∪ StrictFpIntefacePathSet**

24- StrictFpInterfaceNotAbstractStaticMethodSet = InterfaceMethodModifier x ParamStore x <elem identifier> **x JavaTypeSet**

- **Ac.modifierMethInterface = Ac.modifier** \{protected, friendly, private}
- **AbStDeModifier** = {staticNotDefaultNotAbstract}
- **SfAbModifierBase = AbStDeModifier x SfModifier**
- **SfAbModifierRestrict = SfAbModifierBase \{(** defaultNotStaticNotAbstract, notStrictfp**),(** staticNotDefaultNotAbstract, notStrictfp**)}**
  "Nous ne pouvons pas avoir de méthodes default non-strictfp ou static non-strictfp quand l'interface englobante est strictfp"
- **InterfaceMethodModifier = Ac.modifierMethInterface x SfAbModifierRestrict**

StrictFpInterfaceNotAbstractStaticMethodPathSet = SIP x StrictFpInterfaceNotAbstractStaticMethodSet

- **SIP = ClassStrictfpInnerInterfacePathSet ∪ InterfaceStrictfpInnerInterfacePathSet ∪ StrictFpInterfacePathSet**

25- StrictFpInterfaceNotAbstractNotStaticMethodSet = InterfaceMethodModifier x ParamStore x <elem identifier> **x JavaTypeSet**

- **Ac.modifierMethInterface = Ac.modifier** \{protected, friendly, private}
- **AbStDeModifier** = {defaultNotStaticNotAbstract}
- **SfAbModifierBase = AbStDeModifier x SfModifier**
- **SfAbModifierRestrict = SfAbModifierBase \{(** defaultNotStaticNotAbstract, notStrictfp**),(** staticNotDefaultNotAbstract, notStrictfp**)}**
  "Nous ne pouvons pas avoir de méthodes default non-strictfp ou static non-strictfp quand l'interface englobante est strictfp"
- **InterfaceMethodModifier = Ac.modifierMethInterface x SfAbModifierRestrict**

StrictFpInterfaceNotAbstractNotStaticMethodPathSet = SIP x StrictFpInterfaceNotAbstractNotStaticMethodSet

- **SIP = ClassStrictfpInnerInterfacePathSet ∪ InterfaceStrictfpInnerInterfacePathSet ∪ StrictFpInterfacePathSet**

26- NotStrictFpInterfaceAbstractMethodSet = InterfaceMethodModifier x ParamStore x <elem identifier> x JavaTypeSet

- **Ac.modifierMethInterface = Ac.modifier \ {protected, friendly, private}**
- **AbStDeModifier** = {abstractNotStaticNotdefault}
- **SfAbModifierBase = AbStDeModifier x SfModifier**
- **SfAbModifierRestrict = SfAbModifierBase \ { (** abstractNotStaticNotdefault, strictfp **) }**
- **InterfaceMethodModifier = Ac.modifierMethInterface x SfAbModifierRestrict**

NotStrictFpInterfaceAbstractMethodPathSet = NSIP x NotStrictFpInterfaceAbstractMethodSet

- **NSIP = InterfaceNotStrictfpInnerInterfacePathSet ∪ ClassNotStrictfpInnerInterfacePathSet ∪ NotStrictFpIntefacePathSet**

27- NotStrictFpInterfaceNotAbstractStaticMethodSet = InterfaceMethodModifier x ParamStore x <elem identifier> x JavaTypeSet

- **Ac.modifierMethInterface = Ac.modifier \ {protected, friendly, private}**
- **AbStDeModifier** = {staticNotDefaultNotAbstract}
- **SfAbModifierBase = AbStDeModifier x SfModifier**
- **InterfaceMethodModifier = Ac.modifierMethInterface x SfAbModifierBase**

NotStrictFpInterfaceNotAbstractMethodPathSet = NSIP x NotStrictFpInterfaceNotAbstractStaticMethodSet

- **NSIP = InterfaceNotStrictfpInnerInterfacePathSet ∪ ClassNotStrictfpInnerInterfacePathSet ∪ NotStrictFpIntefacePathSet**

28- NotStrictFpInterfaceNotAbstractNotStaticMethodSet = InterfaceMethodModifier x ParamStore x <elem identifier> x JavaTypeSet

- **Ac.modifierMethInterface = Ac.modifier \ {protected, friendly, private}**
- **AbStDeModifier = {defaultNotStaticNotAbstract}**
- **SfAbModifierBase = AbStDeModifier x SfModifier**
- **InterfaceMethodModifier = Ac.modifierMethInterface x SfAbModifierBase**

NotStrictFpInterfaceNotAbstractNotStaticMethodPathSet = NSIP x NotStrictFpInterfaceNotAbstractNotStaticMethodSet

- **NSIP = InterfaceNotStrictfpInnerInterfacePathSet ∪ ClassNotStrictfpInnerInterfacePathSet ∪ NotStrictFpIntefacePathSet**

FieldSet = InterfaceFieldSet ∪ BlockFinalFieldSet ∪ BlockNotFinalFieldSet ∪ ClassFinalStaticFieldSet ∪ ClassNotFinalStaticFieldSet ∪ ClassFinalNotSaticFieldSet ∪ ClassNotFinalNotStaticFieldSet

FieldPathSet = BlockFieldPathSet ∪ ClassFieldPathSet ∪ InterfaceFieldPathSet

FieldPath = InterfaceFieldPath ∪ ClassFieldPath ∪ BlockFieldPath

FinalFieldPathSet = InterfaceFieldPathSet ∪ BlockFinalFieldPathSet ∪ ClassFinalStaticFieldPathSet ∪ ClassFinalNotSaticFieldPathSet

NotFinalFieldPathSet = BlockNotFinalFieldSet ∪ ClassNotFinalStaticFieldSet ∪ ClassNotFinalNotStaticFieldSet

StaticFieldPathSet = InterfaceFieldPathSet ∪ ClassNotFinalStaticFieldPathSet ∪ ClassFinalStaticFieldPathSet

NotSaticFieldPathSet = BlockFinalFieldPathSet ∪ BlockNotFinalFieldPathSet ∪ ClassFinalNotSaticFieldPathSet ∪ ClassNotFinalNotStaticFieldSet

InterfaceFieldPath = InterfacePathSet ∪ InnerInterfacePathSet

ClassFieldPath = ClassPathSet ∪ InnerClassPathSet

BlockFieldPath = ConstructorPathSet ∪ NotAbstractMethodePathSet ∪ BlockPathSet

InterfaceFieldPathSet = InterfaceFieldPath x InterfaceFieldSet

ClassFieldPathSet = ClassFinalStaticFieldPathSet ∪ ClassFinalNotStaticFieldPathSet ∪ ClassNotFinalStaticFieldPathSet ∪ ClassNotFinalNotStaticFieldPathSet

BlockFieldPathSet = BlockFinalFieldPathSet ∪ BlockNotFinalFieldPathSet

1-InterfaceFieldSet = FieldModifier x <elem identifier>

- **Ac.modifierRest = Ac.modifier** \{protected, friendly, private}
- **Af.modifierRest = Af.modifier** \{ abstractNotFinal, notAbstractNotFinal }
- **St.modifierRest = St.modifier** \{notstatic}
- **FieldModifier = Ac.modifierRest x St.modifierRest x Af.modifierRest**

2-BlockFinalFieldSet = FieldModifier x <elem identifier>

- **Af.modifierRest = Af.modifier** \{ abstractNotFinal, notAbstractNotFinal }
- **FieldModifier = Af.modifierRest**

BlockFinalFieldPathSet = BlockFieldPath x BlockFinalFieldSet

3-BlockNotFinalFieldSet = FieldModifier x <elem identifier>

- **Af.modifierRest = Af.modifier** \{ abstractNotFinal, finalNotAbstract}
- **FieldModifier = Af.modifierRest**

BlockNotFinalFieldPathSet = BlockFieldPath x BlockNotFinalFieldSet

4-ClassFinalStaticFieldSet = FieldModifier x <elem identifier>

- **Af.modifierRest = Af.modifier** \{ abstractNotFinal, notAbstractNotFinal }
- **AfVolModifierRest = (Af.modifierRest x Vo.modifier)** **{ (** volatile, finalNotAbstract **) }**
- **St.modifierRest = St.modifier** \{notStatic}
- **FieldModifier = Ac.modifier x St.modifierRest x AfVolModifierRest x Tr.modifier**

ClassFinalStaticFieldPathSet = ClassFieldPath x ClassFinalStaticFieldSet

5-ClassFinalNotStaticFieldSet = FieldModifier x <elem identifier>

- **Af.modifierRest = Af.modifier** \{ abstractNotFinal, notAbstractNotFinal }
- **AfVolModifierRest = (Af.modifierRest x Vo.modifier)** **{ (** volatile, finalNotAbstract **) }**
- **St.modifierRest = St.modifier** \{static}
- **FieldModifier = Ac.modifier x St.modifierRest x AfVolModifierRest x Tr.modifier**

ClassFinalNotStaticFieldPathSet = ClassFieldPath x ClassFinalNotStaticFieldSet

6-ClassNotFinalStaticFieldSet = FieldModifier x <elem identifier>

- **Af.modifierRest = Af.modifier** \{ abstractNotFinal, finalNotAbstract }
- **AfVolModifierRest = (Af.modifierRest x Vo.modifier)**
- **St.modifierRest = St.modifier** \{notStatic}
- **FieldModifier = Ac.modifier x St.modifierRest x AfVolModifierRest x Tr.modifier**

ClassNotFinalStaticFieldPathSet = ClassFieldPath x ClassNotFinalStaticFieldSet

7-ClassNotFinalNotStaticFieldSet = FieldModifier x <elem identifier>

- **Af.modifierRest = Af.modifier** \{ abstractNotFinal, finalNotAbstract }
- **AfVolModifierRest = (Af.modifierRest x Vo.modifier)**
- **St.modifierRest = St.modifier** \{static}
- **FieldModifier = Ac.modifier x St.modifierRest x AfVolModifierRest x Tr.modifier**

ClassNotFinalNotStaticFieldPathSet = ClassFieldPath x ClassNotFinalNotStaticFieldSet

---

1-InterfaceExtendedModelSet = (InterfaceUnionPathSet x {extendTo} x InterfaceUnionPathSet) ⊕ ((InterfaceUnionPathSet x {extendTo}) x InterfaceExtendedModelSet)

- **InterfaceUnionPathSet = InterfacePathSet ∪ InnerInterfacePathSet**

ClassChildModelSet = ClassExtendedModelSet ∪ ClassImplementModelSet

1-ClassExtendedModelSet = (NotFinalClassUnionPathSet x {extendTo} x AllClassUnionPathSet) ⊕ ((NotFinalClassUnionPathSet x {extendTo}) x ClassExtendedSet)

- **AllClassUnionPathSet = ClassPathSet ∪ InnerClassUnionPathSet**
- **NotFinalClassUnionPathSet = NotFinalClass ∪ NotFinalInnerClassSet**

2-ClassImplementModelSet = InterfaceUnionPathSet x {implementTo} x AllClassUnionPathSet

- **AllClassUnionPathSet = ClassPathSet ∪ InnerClassUnionPathSet**
- **InterfaceUnionPathSet = InterfacePathSet ∪ InnerInterfacePathSet**

1-ElemSet = PackageSet ∪ ClassSet ∪ InnerClassSet ∪ InterfaceSet ∪ InnerInterfaceSet ∪ BlockSet ∪ ConstructorSet ∪ MethodeSet ∪ FieldSet

ElemPathSet = PackagePathSet ∪ ClassPathSet ∪ InnerClassPathSet ∪ InterfacePathSet ∪ InnerInterfacePathSet ∪ BlockPathSet ∪ ConstructorPathSet ∪ MethodePathSet ∪ FieldPathSet

---

$\mathbb{N}$ *(Nombres entiers naturels)*

$\mathbb{N}^*$ *(Nombres entiers naturels sans le zéro)*

$\mathbf{Z}$ *(Nombres entiers relatifs)*

$\mathbb{R}$ *(Nombres réels)*

Bool = {true, false}

Char = {Unicode input character} \ { " , \}

String = {null} $\oplus$ (**Char** x **Sring**)

JavaPrimTypeValue = $\mathbf{Z} \cup \mathbb{R} \cup$ **Bool** $\cup$ **String**

JavaTypeValue = **JavaPrimTypeValue** $\cup$ **JavaType** $\cup$ {null}

ParamStoreValue = { $\mathbb{N}^* \longrightarrow$ **JavaTypeValue** }

StartTimeStamp, EndTimeStamp, TimeStamp = $\mathbb{N}^*$ *(Représente une empreinte de temps)*

ThreadId = $\mathbb{N}^*$

InstanceId = $\mathbb{N}^* \cup$ {notInstance} **(Remarque : Préoccupation technique : nous ne pouvons pas garantir que le hashCode d'une instance soit unique)** *"Dans le cadre de notre approche 'top → bottom', nous faisons abstraction des préoccupations techniques pour fournir une*

fonctionnalité utile donnant le droit de comparer les instances. Nous émettons l'hypothèse que nous avons accès à un identifiant d'instance unique"

Remarque : Une méthode héritée d'une classe/interface va être appelée au travers (d'une instance ou non) d'une classe fille. Il faut pouvoir connaître le type de la classe fille.

EventsSet = ConstructorCallSet ∪ MethodCallSet ∪ BlockCallSet ∪ ThreadCallSet ∪ ReadFieldSet ∪ WriteFieldSet ∪ CatchExceptionSet ∪ ThrowExceptionSet

DynSet = ConstructCallDynSet ∪ NotStaticMethodCallDynSet ∪ StaticMethodCallDynSet ∪ NotStaticBlockDynSet ∪ StaticBlockDynSet ∪ ThreadDynSet ∪ NotStaticReadFieldDynSet ∪ StaticReadFieldDynSet ∪ StaticWriteFieldDynSet ∪ NotSaticWriteFieldDynSet ∪ ThrowExceptionDynSet ∪ CatchExceptionDynSet

MethodCallSet = NotStaticMethodCallSet ∪ StaticMethodCallSet

BlockCallSet = NotStaticBlockCallSet ∪ StaticBlockCallSet

ReadFieldSet = ReadNotStaticFieldSet ∪ ReadStaticFieldSet

WriteFieldSet = WriteStaticFieldSet ∪ WriteNotSaticFieldSet

1-ConstructorCallSet = BaseConstructorCallSet ⊕ (BaseConstructorCallSet x EventsSet$^{(5)}$)

- **AllNotAbstractClassPathSet = InnerNotAbstractClassPathSet ∪ NotAbstractClassPathSet**
- **InstanceIdRest = InstanceId \ {notInstance}**
- **InstanceIdEffType = InstanceIdRest X AllNotAbstractClassPathSet** $^{(1)}$
- **ConstructCallDynSet = InstanceIdEffType** $^{(2)}$ **x StartTimeStamp x EndTimeStamp x ThreadId x ParamStoreValue**
- **BaseConstructorCallSet = ConstructCallDynSet x ConstructorPathSet**

BaseMethodCallSet = BaseNotStaticMethodCallSet ∪ BaseStaticMethodCallSet

2-NotStaticMethodCallSet = (BaseNotStaticMethodCallSet) ⊕ (BaseNotStaticMethodCallSet x EventsSetRest$^{(5)}$)

- **AllNotAbstractClassPathSet = InnerNotAbstractClassPathSet ∪ NotAbstractClassPathSet**
- **InstanceIdRest = InstanceId \ {notInstance}**
- **InstanceIdEffType = InstanceIdRest X AllNotAbstractClassPathSet** $^{(1)}$
- **NotStaticMethodCallDynSet = InstanceIdEffType x StartTimeStamp x EndTimeStamp x ThreadId x ParamStoreValue x JavaTypeValue**
- **ElemCalledPathSet = NotAbstractNotStaticMethodPathSet**
- **EventsSetRest = EventsSet \ BlockCallSet**
- **BaseNotStaticMethodCallSet = NotStaticMethodCallDynSet x ElemCalledPathSet**

3-StaticMethodCallSet = BaseStaticMethodCallSet ⊕ (BaseStaticMethodCallSet x EventsSetRest[5])

- InstanceIdRest = InstanceId \ $\mathbb{N}^*$
- AllClassInterfacePathSet = ClassPathSet ∪ InnerClassPathSet ∪ InterfacePathSet ∪ InnerInterfacePathSet
- InstanceIdEffType = InstanceIdRest X AllClassInterfacePathSet [1]
- StaticMethodCallDynSet = InstanceIdEffType x StartTimeStamp x EndTimeStamp x ThreadId x ParamStoreValue x JavaTypeValue
- EventsSetRest = EventsSet \ NotSaticBlockCallSet
- BaseStaticMethodCallSet = StaticMethodCallDynSet x NotAbstractStaticMethodPathSet

BaseBlockCallSet = BaseNotStaticBlockCallSet ∪ BaseStaticBlockCallSet

4-NotStaticBlockCallSet = BaseNotStaticBlockCallSet ⊕ (BaseNotStaticBlockCallSet x EventsSetRest[5])

- InstanceIdRest = InstanceId \ {notInstance}
- NotStaticBlockDynSet = InstanceIdRest[2] x StartTimeStamp x EndTimeStamp x ThreadId
- EventsSetRest = EventsSet \ SaticBlockCallSet
- BaseNotStaticBlockCallSet = NotStaticBlockDynSet x NotStaticBlockPathSet

5-StaticBlockCallSet = BaseStaticBlockCallSet ⊕ (BaseStaticBlockCallSet x EventsSet)

- StaticBlockDynSet = StartTimeStamp x EndTimeStamp x ThreadId
- BaseStaticBlockCallSet = StaticBlockDynSet x StaticBlockPathSet

6-ThreadCallSet = BaseThreadCallSet ⊕(BaseThreadCallSet x EventsSetRest[5])

- **AllNotAbstractClassPathSet = InnerNotAbstractClassPathSet ∪ NotAbstractClassPathSet**
- **InstanceIdRest = InstanceId \ {notInstance}**
- **InstanceIdEffType = InstanceIdRest X AllNotAbstractClassPathSet [1]**
- **ThreadDynSet = InstanceIdEffType x StartTimeStamp x EndTimeStamp x ThreadId**
- **EventsSetRest \ BlockCallSet**
- **BaseThreadCallSet = NotStaticMethodCallDynSet x NotStaticMethodPathSet[3] x ThreadDynSet x NotStaticMethodPathSet [3]**

BaseReadFieldSet = ReadNotStaticFieldSet ∪ BaseReadStaticFieldSet

7-ReadNotStaticFieldSet = NotStaticReadFieldDynSet x NotSaticFieldPathSet

- **AllNotAbstractClassPathSet = InnerNotAbstractClassPathSet ∪ NotAbstractClassPathSet**
- **InstanceIdRest = InstanceId \ {notInstance}**
- **InstanceIdEffType = InstanceIdRest X AllNotAbstractClassPathSet**
- **NotStaticReadFieldDynSet = InstanceIdEffType x TimeStamp x JavaTypeValue**

8-ReadStaticFieldSet = BaseReadStaticFieldSet ⊕ (BaseReadStaticFieldSet x StaticBlockCallSet)

- InstanceIdRest = InstanceId \ $\mathbb{N}^*$
- AllClassInterfacePathSet = ClassPathSet ∪ InnerClassPathSet ∪ InterfacePathSet ∪ InnerInterfacePathSet
- InstanceIdEffType = InstanceIdRest X AllClassInterfacePathSet
- StaticReadFieldDynSet = InstanceIdEffType x TimeStamp x JavaTypeValue
- BaseReadStaticFieldSet = StaticReadFieldDynSet x FieldPathSet

BaseWriteFieldSet = BaseWriteStaticFieldSet ∪ WriteNotSaticFieldSet

9-WriteStaticFieldSet = BaseWriteStaticFieldSet ⊕ (BaseWriteStaticFieldSet x StaticBlockCallSet)

- InstanceIdRest = InstanceId \ $\mathbb{N}^*$
- AllClassInterfacePathSet = ClassPathSet ∪ InnerClassPathSet ∪ InterfacePathSet ∪ InnerInterfacePathSet
- InstanceIdEffType = InstanceIdRest X AllClassInterfacePathSet
- StaticWriteFieldDynSet = InstanceIdEffType x TimeStamp x JavaTypeValue x JavaTypeValue
- StaticNotFinalFieldPathSet = NotFinalFieldPathSet ∩ StaticFieldPathSet
- BaseWriteStaticFieldSet = StaticWriteFieldDynSet x StaticNotFinalFieldPathSet

10-WriteNotSaticFieldSet = NotSaticWriteFieldDynSet x StaticNotFinalFieldPathSet

- **AllNotAbstractClassPathSet = InnerNotAbstractClassPathSet ∪ NotAbstractClassPathSet**
- **InstanceIdRest = InstanceId \ {notInstance}**
- **InstanceIdEffType = InstanceIdRest X AllNotAbstractClassPathSet**
- **NotSaticWriteFieldDynSet = InstanceIdEffType x TimeStamp x JavaTypeValue x JavaTypeValue**
- **StaticNotFinalFieldPathSet = NotFinalFieldPathSet ∩ StaticFieldPathSet**

11-(Base)ThrowExceptionSet = ThrowExceptionDynSet x AllNotAbstractClassPathSet

- **AllNotAbstractClassPathSet = InnerNotAbstractClassPathSet ∪ NotAbstractClassPathSet**
- **JavaTypeValueRest = JavaTypeValue \ JavaPrimTypeValue**
- **ThrowExceptionDynSet = InstanceIdRest x TimeStamp x JavaTypeValueRest**

12-(Base)CatchExceptionSet = CatchExceptionDynSet x AllNotAbstractClassPathSet

- **AllNotAbstractClassPathSet = InnerNotAbstractClassPathSet ∪ NotAbstractClassPathSet**
- **JavaTypeValueRest = JavaTypeValue \ JavaPrimTypeValue**
- **CatchExceptionDynSet = InstanceIdRest x TimeStamp x JavaTypeValueRest**

13-ProgramEntryCallSet = BaseProgramEntryCallSet ⊕ (BaseProgramEntryCallSet x EventsSetRest[5])

- InstanceIdRest = InstanceId \ $\mathbb{N}^*$
- AllNotAbstractClassPathSet = InnerNotAbstractClassPathSet ∪ NotAbstractClassPathSet
- InstanceIdEffType = InstanceIdRest X AllNotAbstractClassPathSet[1]
- StaticMethodCallDynSet = InstanceIdEffType x StartTimeStamp x EndTimeStamp x ThreadId x ParamStoreValue x JavaTypeValue
- EventsSetRest = EventsSet \ NotSaticBlockCallSet
- BaseProgramEntryCallSet = StaticMethodCallDynSet[4] x NotAbstractStaticMethodPathSet

[1] (Pour tous types effectifs étant égales au type de la classe ou se trouve la (méthode/constructeur/champ) appelée/accédée ou pour tous types effectifs se trouvant dans un modèle d'héritage où le type effectif est le fils (direct ou pas) du type de la classe qui comporte la méthode appelée)

[2] (Pas d'instance encore créée, mais une instance est en cour de création)

[3] (La méthode appelée par le système est la méthode 'run' de la classe 'Thread' suite à un appel d'une méthode 'start')

[4] (Souvant la fonction 'main')

[5] Applicable sous réserve des règles d'accessibilité de Java